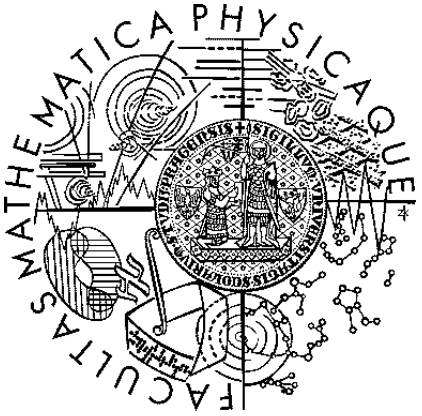
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Petr Poštulka

Online monitoring of electronic trading

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Informatika, obor Softwarové systémy

2010

First of all I wish to express my sincere gratitude and appreciation to my supervisor, Michal Kopecký, for his planning the project of my work, his thoughtful guidance, valuable suggestions, precious comments during discussions, kind supervision with me and his continuous encouragement to make my work successful.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 10.12. Petr Poštulka

Contents

List of Figures

Název práce: Online monitoring of electronic trading
Autor: Petr Poštulka
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.
e-mail vedoucího: Michal.Kopecky@mff.cuni.cz

Abstrakt: Práce analyzuje procesy uvnitř brokerské společnosti a na základě této analýzy popisuje návrh a implementaci řešení, které umožňuje sledování on-line provozu na jednotlivých burzách. Monitorovací systém poskytuje detailní přehled o aktuálních objednávkách a obchodech na jednotlivých burzách, umožňuje vyhledávání a filtrovaní dat, stejně jako automatické vytváření a odesílání reportů a protokolů s informacemi o latencích zpracovávaných zpráv. Systém počítá se zpracováním velkého množství dat, dosahujícího přibližně milionu zpráv za den, ve špičkách pak stovek zpráv za sekundu.

Klíčová slova: elektronické obchodování, monitorování, akcie

Title: Online monitoring of electronic trading
Author: Petr Poštulka
Department: Department of Software Engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: Michal.Kopecky@mff.cuni.cz

Abstract: The thesis analyses the processes within a brokerage firm and based on this analysis describes the design and implementation of a solution, that allows user to monitor the electronic trading infrastructure at real-time. The monitoring system provides a detailed overview of live orders and currently realized transactions on various stock exchanges, allows searching and filtering data, as well as automatic creation and distribution of reports and protocols with information about message latencies. The system design takes into account a large amounts of data, reaching approximately million messages per day and hundreds of messages per second at peak times.

Keywords: electronic trading, monitoring, security

Chapter 1

Introduction

1.1 Definition of electronic trading and its use in financial markets

At first it should be noted, that the thesis contains a lot of terms and abbreviations, which the reader doesn't have to know. In such cases please see the Appendix B, where all these terms are explained.

Historically stock markets provided location where market participants met to trade stocks and derivatives. The New York Stock Exchange (shortly NYSE) is an example of such stock market. It is a representative of old process of exchange trading, where each listed stock has a specialist, who oversees all of the trades for a particular stock and orders are phoned down to a floor trader and done manually in so-called open outcry. With the improvement in communications technology in the late 20th century, the electronic trading, which is sometimes also called *etrading* (shortly ET), started to rapidly gain a ground in financial markets. Common understanding of what actually constitutes electronic trading comprises a wide variety of systems of different complexity, covering the trading process either partially or at a whole. These systems include *electronic order routing*, which ensures delivery of orders from buy side to sell side counter-parties, *automated order matching engines*, which are responsible for matching corresponding buy and sell orders and *electronic transmission of market data*, that is important both for *pre-trade analysis* (bid/offer quotes and market depths) and for *post-trade processing* (execution prices and volumes) [Cgfs01].

The main advantage of electronic trading is that there is no need for physical locations where counter-parties met and negotiate, because all transactions can be completed remotely through electronic trading platforms. The most famous electronic equity market is NASDAQ, which is not a physical entity, but a network between thousands of computers. With this new approach in financial markets, old process of exchange trading known as open outcry has practically disappeared. Also the NYSE was influenced by electronic trading in a form of electronic system called SuperDOT, which was designed for small order entry on this market. But remote access was not the only advantage that was brought about by ET introduction. Other equally important benefits are:

- **Increased operation efficiency** - It is much easier to connect front office, middle office and back office systems and pass trades straight through all three parts. This type of linking execution, confirmation, clearing and settlement is known as Straight-Through-Processing (STP) and brings huge efficiency boost.
- **Reduced cost of transactions** - By automating as much of the trading process as possible, costs are brought down. The goal is to reduce the incremental cost of trades as close to zero as possible, so that

increased trading volumes don't lead to significantly increased costs. This has translated to lower costs for investors [Et09].

- **Minimised the risk of errors** - With STP there is no need for intermediate manual intervention, so the trade reporting and record keeping is less error prone.
- **Greater liquidity** - Because there is no need to concentrate all trading participants in one place, it allows different companies to trade with one another. With increasing number of buyers and sellers markets become more liquid.
- **Increased transparency** - Because electronic trading systems capture all important trading information automatically, prices are more easily accessible and more quickly available. This gives greater extent of transparency than in traditional trading.

Thanks to the above-mentioned advantages, which are only the tip of the iceberg, many investment firms are increasing their spending on technology electronic trading, which is certainly the future of financial markets. Because usually there are many systems, which form a whole electronic trading infrastructure, the more important is to have an overview of whole trading process. It is crucial to get different types of information in timely manner - e.g. which orders are open on which markets, what volume was traded on a particular stock, at which system in the trading chain the order got stuck and much more. All these information help employees of brokerage firm to be more productive, to make less errors during their work and to provide clients with the best possible support.

1.2 Stock market participants

Before we describe the concrete targets of this thesis, it is important to introduce the main stock market participants and their role in trading process, so the reader gets a better picture about the trading process and why the monitoring system is so important for a brokerage firm. The stock market is comprised of people. People running the stock exchanges, people running stock brokerages and, of course, people buying and selling stocks. While there are other important participants, the following groups are the basic and key players in making the stock market function properly.

- The companies issuing shares of stock
- The stock exchanges listing the shares of stock
- The stock brokers providing buyers and sellers access to the exchanges
- The buyers and the sellers trading the stock

The interaction between market participants is schematically outlined in Figure 1.1.

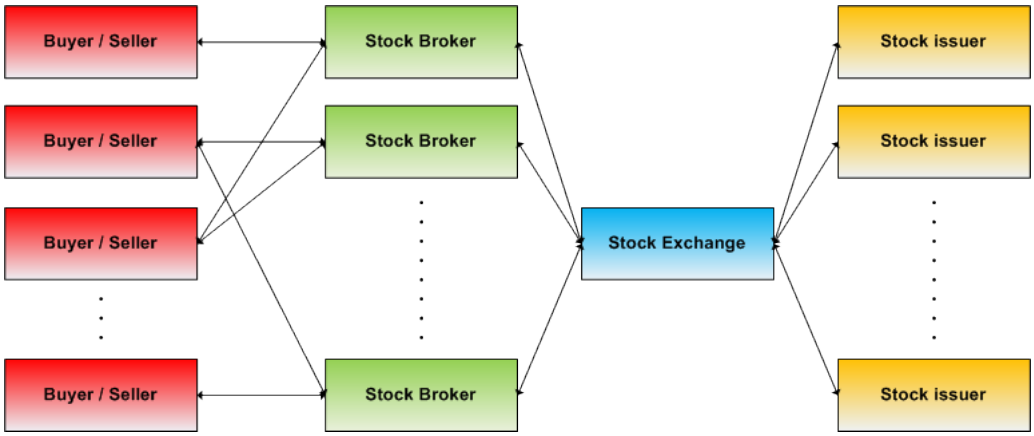


Figure 1.1: The interaction between various stock market participants

From the figure is clear that stock brokers play a significant role in stock markets and are the main link between sellers and buyers (e.i brokerage firm's clients), and exchanges. Now the question is what is the reason for the existence of brokerage firms? The answer is simple. It makes lives easier for its clients. To trade on the stock market, you must be a member of this market and in many cases it is not an easy procedure to become a member. You must pass the brokerage exams, participate in guarantee funds, pay memberhip fees, and a lot of administrative tasks must be done, which, of course, involves considerable expenses.

There are many brokerage firms around. A lot of them are bigger entities like banks. One of the biggest is Barclays in United Kingdom, but there are also other independent smaller stock brokers that specialize in just stock broking. Each broker provides some level of services. There are full-service brokers, that can provide services such as trade execution, stock selection advise, retirement advise or insurance and on the other hand there are discount brokers that offer only the tools and services you need to place trades as an individual investor. Logically full-service brokers typically charge higher commissions than discount brokers. It is common, that brokerage firms are members of several stock markets and provide access to them to many clients, where by client is meant both large entities like banks and funds or even other brokerage firms and small individual investors from retail sector. Each broker may take advantage of its membership on stock markets and can trade securities on its own account. This type of broker is called broker-dealer, because it is acting like a broker when trading on behalf of its customers and is said to be acting as dealer when executing trades for its own account. In the Figure 1.2 you can see an example of broker-dealer, which is a member of three exchanges and provides the access to them to two other brokers and to a group of retail investors.

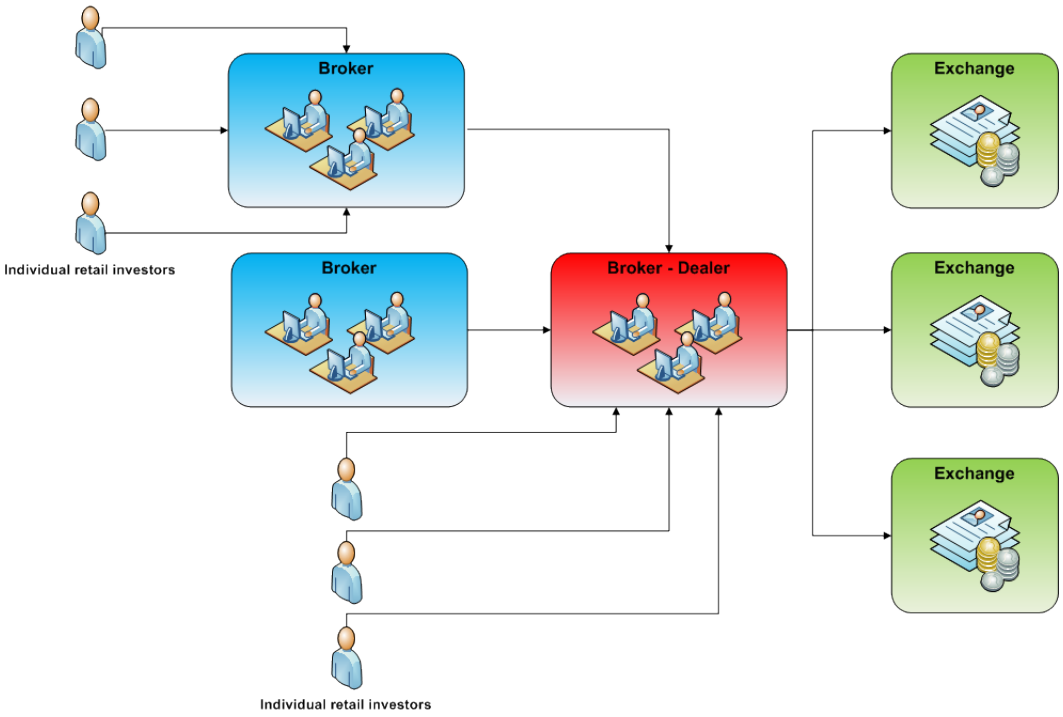


Figure 1.2: Role of broker-dealer in stock markets

If we look at the trade execution service in more detail, most brokers provide their clients various ways how to route their orders. Main examples are:

- **By phone to a sales trader** - Despite the exponential rise in electronic trading there are still huge volumes of trades that are done via plain old telephone service.
- **Electronically to a sales trader** - When orders are routed with this way, they are called *care orders*. Order is electronically routed through some kind of protocol to the trading terminal where usually sits a sales trader, who is responsible for further order processing.
- **Electronically to an algorithmic system** - Orders are automatically processed by various types of algorithmic systems, which implement various trading strategies without human intervention.
- **Electronically and directly to a market** - This type of routing is known as *direct market access* (shortly DMA) and becomes very popular with introduction of electronic trading.

Each type of routing has its advantages and disadvantages and it is on client to choose, which type he wants to use. It is common that orders accepted by phone or care orders are charged with higher commissions than DMA orders because there is a need of human intervention and the sales trader must decide how to process the incoming order. He can choose to route it to a market, to let the algorithmic engine process this order on behalf of him and many other possible scenarios. The only criterion, which the brokerage firm should guarantee, no matter of which type of routing the client choose, is the best execution for all trades.

The whole process of trade execution is done at front office desk. But there are other important departments, which make brokerage firm fully functional institution. Each brokerage firm consists of, among others, the following departments:

- **Front office** - A part of brokerage firm that is "facing the clients". It is responsible for fulfilment of client orders and consists mainly of sales staff, traders and also from a client support desk. The monitoring system is mainly intended for this department, especially for a client support team. They have to know what open orders have each client on individual stock exchanges, so whenever there are some troubles such as connectivity issue, they know which orders have to be cancelled. They also need an overview about what volumes have each client traded on particular stocks, what are the average prices on given orders and number other important information, so they can give the client precise information at any point of time and assure the best care.
- **Middle office** - Department responsible for position-keeping, risk management, trade confirmation and P&L calculations. Usually it is also a mediator between front office and back office teams. For middle office department, the monitoring system is important especially because of trade confirmation management and revenues and costs calculation.
- **Back office** - Department where physical settlement of all transactions is done. Transfer of securities and money and the tracking of "failure to deliver" are handled there also. The usage of monitoring system within this department is useful for checking what particular clients have traded. This gives the back office members an overview about what transactions should be settled with each client and if there are any clients, who fail to deliver the required transactions.

From above paragraphs it is clear, that the monitoring system is a very important part of each trading infrastructure and that it simplifies the work of major departments in a brokerage firm.

1.3 Aim of this Work

The goal of this work is to analyse processes in a brokerage firm and based on this analysis to design and implement a system for online monitoring of electronic trading infrastructure. The importance should be put on:

- **Robustness** - System must have a high degree of stability and be resistant to invalid inputs into the system.
- **Modularity** - System should be modular designed and documented so that it will be easy to:
  - Add new trading systems, so they can be monitored without major changes of system implementation.
  - Extend the functionality of the system to add new monitoring features.
  - Add new markets and monitor order states and traded volumes.
  - Add new clients and monitor their traffic.
- **Performance** - System should be prepared to handle increasing volumes of data, which means:
  - Process approximately million of messages per day across the whole trading infrastructure.
  - Process messages in real-time with the delay not exceeding several tens of seconds.

1.4 Structure of this Document

The next Chapter 2 describes the messaging standard developed specifically for the real-time electronic exchange of securities transactions. It mentions most widely used messages and explains basic functionality of the protocol.

Chapter 3 then analyses system requirements, outlines existing trading infrastructure with its current possibilities and shortcomings in terms of monitoring in thesis submitting company, further in the thesis denoted as sponsoring company, and defines specific requirements. Chapter also briefly describes available monitoring systems and discusses their suitability for deployment at sponsoring company.

Chapter 4 describes several different alternative architectures of system implementation and their main characteristics and suitability for monitoring system.

The following Chapter 5 discusses different types of communication middleware, their pros and cons from the monitoring system point of view and based on findings chooses the most suitable middleware for system implementation.

Chapter 6 briefly describes several plug-in frameworks and chooses the most suitable one for providing the modular design of whole system.

Chapter 7 then explains advantages of application framework implementation, describes its architecture and technical details and provides an overview about what general functionality of monitoring system it encapsulates.

Chapter 8 includes technical details about monitoring system architecture, its integration into application framework described in previous Chapter 7 and its implementation, including description of restrictions the system has and decisions made during the development.

Final Chapter 9 recapitulates whether the implemented system meets all targets specified in Chapter 3 and suggests the enhancements for future work.

Chapter 2

Financial Information eXchange Protocol

2.1 Introduction

The Financial Information eXchange (FIX) effort was initiated in 1992 by a group of institutions and brokers interested in streamlining their trading processes. These firms felt that not only they, but the industry as a whole, could benefit from efficiencies derived through the electronic communication of indications, orders and executions. The result is FIX, an open message standard controlled by no single entity, that can be structured to match the business requirements of each firm [FIXProtocolLtd01].

Although FIX protocol was created especially for the front-office area, i.e. for pre-trade communication and trade execution, in last years it is expanding to post-trade space to support Straight-Through-Processing including allocations and confirmation. The protocol is very widely spread and all major stock exchanges and investment banks are using it in their systems for electronic trading. In Figure 2.1 you can see an example of FIX protocol integration in financial markets.

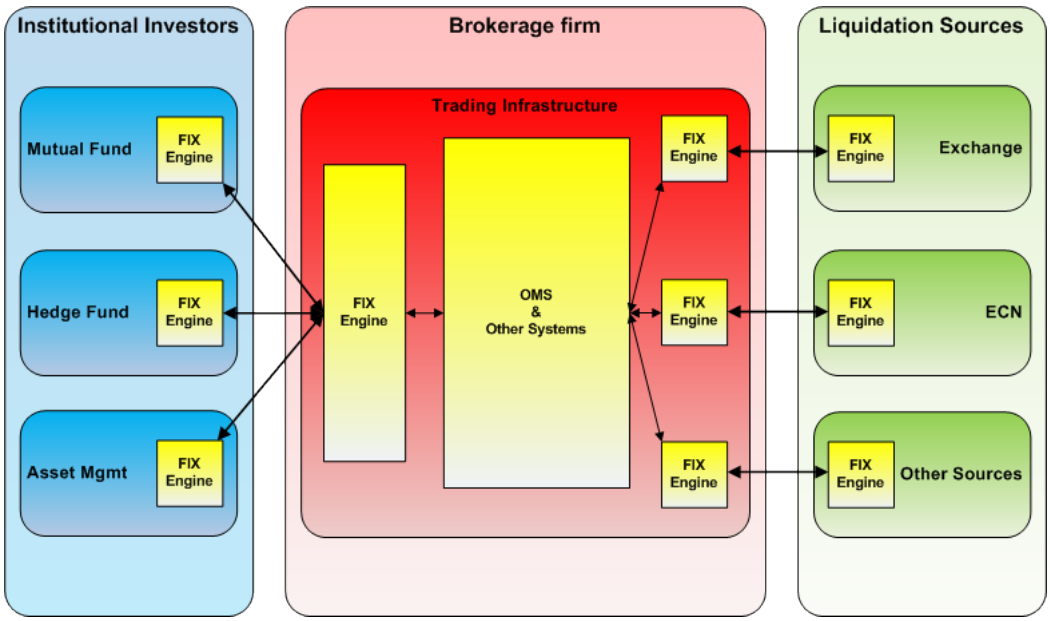


Figure 2.1: FIX protocol integration in financial markets

Among the biggest benefits of the protocol belongs [FIXProtocolLtd03]:

- FIX is an open and free protocol specification, around which software developers can create commercial or open-source software, as they see fit.
- Protocol is supported globally by several committees and working groups.
- Represents a flexible and growing platform that survived the test of time.
- Represents an industry standard protocol, which improves system scalability, operational efficiency and control of the investment process through automation.
- Reduces the need to support multiple solutions across business units, which is inefficient and not cost-effective.
- Helps to reduce costs associated with manual errors in the trade entry, execution, and post-trade processes by allowing for a more seamless integration of various order management applications and functionality.
- Eliminates the need of voice-based telephone conversations and as a consequence reduces the time spent to realize the transaction.

With new requirements coming from users of the protocol, new versions were released and the expansion of the protocol will definitely continue in the future. The most current version is FIX 5.0 Service Pack 2 released in April 2009, but all previous versions are still supported and used by many institutions. Because FIX protocol is very complex, only its basic functionality and a small subset of messages, which protocol supports will be mentioned in the following sections. Though this description should suffice for general insight of implemented system, more information about the protocol can be found in [FIXProtocolLtd01].

2.2 FIX message format

As it was mentioned in previous section, FIX is a message-based protocol. Each message consists of a set of fields which are represented as tag-value pairing. Tag is a string representation of an integer, which identifies the meaning of the field and value is a specific data type mapped to a string representation indicating the field value. The individual fields are separated from each other by a SOH delimiter, which is described as "Start of Heading" and it is the 2nd character of the first 32 non-printing ASCII characters set and is allocated the value 001. FIX message is logically divided into three sections:

- **Header** - Section containing fields for FIX version and message type identification, field with a sequence number of message within a session, a set of routing fields, which are used for routing the message between counterparties and two fields to help with resending messages.
- **Body** - Consists of general fields required for particular message and possibly some optional fields.
- **Trailer** - Section with only one required field, which contains a checksum of message and two optional fields for message signature.

There exists set of mandatory fields for each message type. These fields must be always presented and can be extended by some optional fields to describe the purpose of specific message in more details. Fields within a message can be defined in any sequence, except some basic rules:

- Each message begins with header section, followed by body and trailer sections.
- Each field must remain within a correct section.
- The header section must begin with BeginString(8) field, which indicates, which version of FIX is used, followed by BodyLength(9) field and MsgType(35) field.
- Trailer must end with CheckSum(10) field, which contains a message checksum.

There exist some additional field sequence rules like rules concerning repeating data group fields, but for simplicity they are not mentioned. FIX protocol defines a concrete set of fields, but to provide a maximum flexibility user-defined fields are supported too and are usually used within an internal communication in a single firm, but can be used for inter-firm communication as well. In Example 2.1 you can see a FIX message representing a new order.

Example 2.1 FIX message representing a new order

```
8=FIX.4.1<SOH>9=248<SOH>35=D<SOH>115=ONBEHALF−31<SOH>116=CARE<SOH>34=30<SOH>49=CLIENT−29<SOH>56=RM<SOH> ↩
    43=N<SOH>52=20090725−06:14:28<SOH>55=CZ0008019106<SOH>11=0−EPTL77920101025<SOH>100=PR<SOH>47=P<SOH> ↩
    21=3<SOH>48=CZ0008019106<SOH>22=4<SOH>54=2<SOH>38=2886<SOH>59=0<SOH>44=3925.000000<SOH>40=2<SOH>15= ↩
    CZK<SOH>10=098<SOH>
```

2.3 Communication model

The basic unit of FIX communication is a session, which is defined as a bidirectional stream of ordered messages between participants of communication. Each message sent through the session must follow a sequence, which is specified by sequence numbers included in each message. The FIX protocol assumes ordered delivery of messages between parties. When dealing with message gaps, all messages subsequent to the last message received will be requested. For example if the receiver misses the second of five messages sent, messages 3-5 will be ignored and a *Resend Request* message for messages 2-5 is generated. If more messages (i.e. messages 6 and 7) were received after the *Resend Request* but before the first resent message (i.e. message 2) these messages will be ignored by the receiver and resent by the sender.

Based on who establishes the communication link, the following terms are used to identify the communication parties:

- **Initiator** - Party, which establishes the communication link and initiate a session with *Logon* message.
- **Acceptor** - Receiving party listening on a specific port and responsible for acknowledging or rejecting the *Logon* message in case the message does not meet the agreed requirements of counterparties.

After successful establishment of FIX session, message exchange process begins. During this process administrative (session layer) and especially application messages are sent between Acceptor and Initiator. Once one of the participants of connection decided to end a session, it will send a *Logout* message to its counterparty. See Figure 2.2 outlining an example of communication model between client and broker.

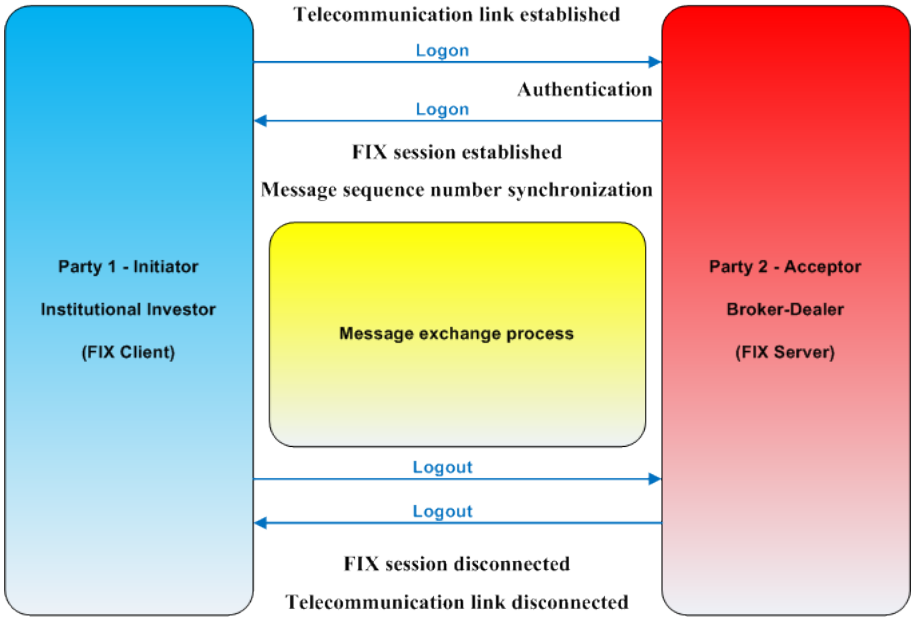


Figure 2.2: FIX protocol communication model

2.4 Basic FIX message types

FIX messages are divided into two logical groups on the basis of what is their purpose. *Session layer* messages include *Logon* and *Logout* messages for initializing and terminating the logical connection together with some other important messages such as:

- ***Heartbeat*** message used for monitoring the status of connection.
- ***TestRequest*** message which enforces counterparty to send a heartbeat message.
- ***ResendRequest*** message used for initiate a retransmission of messages in case of missed messages by receiving application or in case of message gap.
- ***SequenceReset*** message for resetting the sequence number to some specific value generally used in 24/7 scenarios.
- ***Reject*** message sent as a reply to message, which can't be processed by receiving application because of invalid format.

The second group called *Application* messages is further logically divided according to which phase of Straight-Through-Processing individual messages belong to. These are a Pre-Trade, Trade and Post-Trade groups and each of them is further divided. Because the most used messages are *Single General Order Handling* messages, which belong to a Trade group and because monitoring system should provide functionality for monitoring traffic especially of these message types, only these are briefly described in the following paragraph [FIXProtocolLtd01].

- ***New Order - Single*** message is used by institutions wishing to electronically submit securities orders to a broker or exchange for execution. Message provides numerous tags to support all possible information required for order description. As an example of such tag, each order can be submitted with special handling instructions (field *HandlInst* with tag number 21), which refer to how the broker should handle the order and is used to determine if the order is Care or DMA.
- ***Order Cancel/Replace Request (a.k.a. Order Modification Request)*** message used to change the parameters of an existing order. Only a limited number of fields can be changed via this message, for example number of shares and price of the order, but also handling instruction and few others.
- ***Order Cancel Request*** message serves for requesting the cancellation of all of the remaining quantity of an existing order.
- ***Order Cancel Reject*** message, which is issued by the broker upon receipt of an *Order Cancel Request* or *Order Cancel/Replace Request* message which cannot be honoured.
- ***Execution Report*** message is used by broker or stock exchange for various needs like:

  1. confirming the receipt of an order
  2. confirming changes to an existing order (i.e. accept cancel and replace requests)
  3. relaying order status information
  4. relaying fill information on working orders
  5. rejecting orders

  For identification of the purpose of the *Execution Report* message, the *ExecType* field with tag number 150 is used.
- ***Order Status Request*** message is used to request the status of existing order.
- ***Don't Know Trade*** message notifies a trading partner that an electronically received execution has been rejected.

2.5 Message flow scenarios

For a better understanding how the most important messages from *Single General Order Handling* group are used during the communication between trading parties, some simple scenarios are described in the following paragraphs. In each figure is outlined a part of message exchange process related to handling of one order.

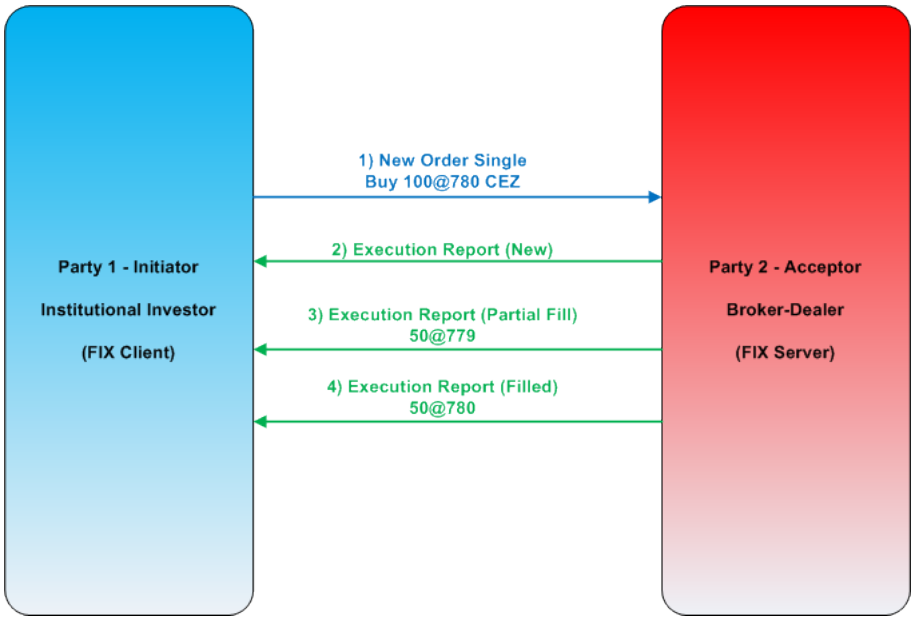


Figure 2.3: Order execution

In Figure 2.3 you can see the message flow related to the simple order execution:

1. Client sends to the broker a buy order for CEZ security with quantity 100 and limit price 780. Limit price indicates that the order must be executed at or below a specified price.
2. Broker sends information to the client that the order was accepted for further processing.
3. Broker sends information to the client that 50 shares of given order were executed at price 779 and order is now partially filled.
4. Broker sends information to the client that another 50 shares of given order were executed at price 780 and order is now fully filled.

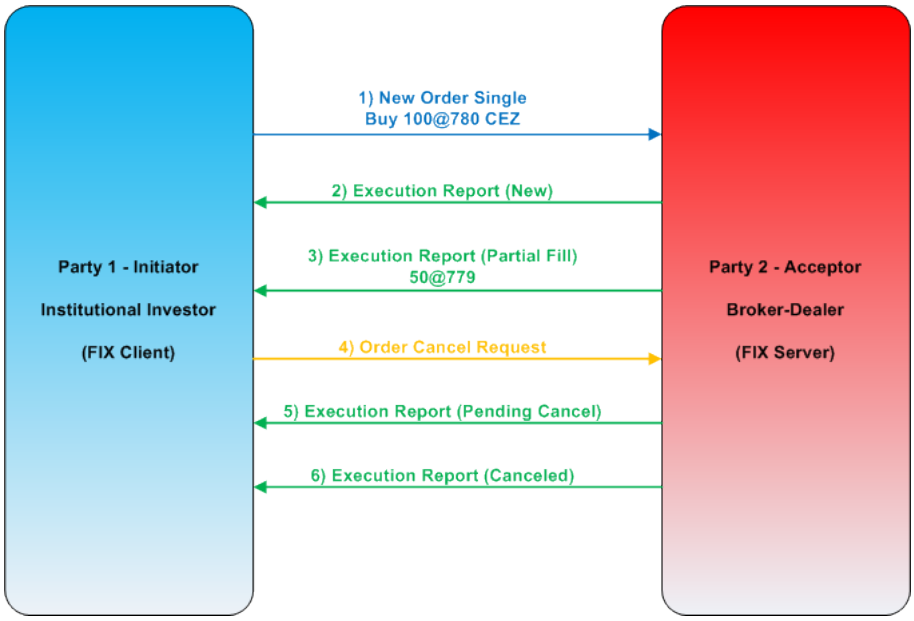


Figure 2.4: Order partially executed and cancelled

Figure 2.4 displays a message flow of order cancellation:

1. Client sends to the broker a buy order for CEZ security with quantity 100 and limit price 780.
2. Broker sends information to the client that the order was accepted for further processing.
3. Broker sends information to the client that 50 shares of given order were executed at price 779 and order is now partially filled.
4. Client requests the cancellation of all of the remaining quantity of an existing order.
5. Broker sends information to the client that the cancel request was received and the cancellation of order is in process.
6. Broker sends information to the client that the remaining quantity of order was successfully cancelled.

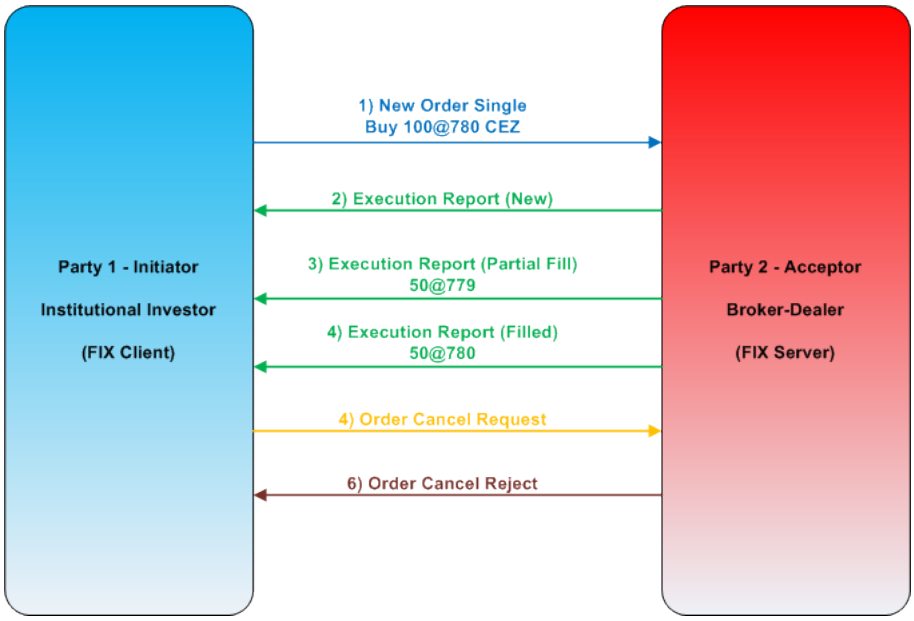


Figure 2.5: Order executed and unsuccessfully cancelled

Figure 2.5 depicts a message flow of unsuccessful cancel request of an already executed order:

1. Client sends to the broker a buy order for CEZ security with quantity 100 and limit price 780.
2. Broker sends information to the client that the order was accepted for further processing.
3. Broker sends information to the client that 50 shares of given order were executed at price 779 and order is now partially filled.
4. Broker sends information to the client that another 50 shares of given order were executed at price 780 and order is now fully filled.
5. Client requests the cancellation of all of the remaining quantity of an existing order.
6. Broker sends information to the client that the cancel request can't be accepted because order is already fully executed.

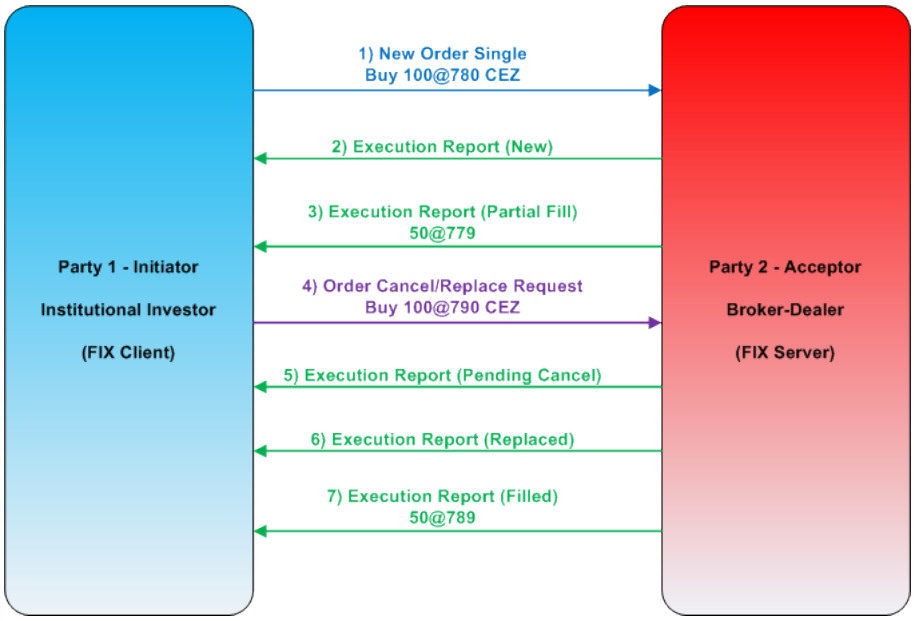


Figure 2.6: Order partially executed, modified and executed with new price

In Figure 2.6 is outlined a message flow of order modification:

1. Client sends to the broker a buy order for CEZ security with quantity 100 and limit price 780.
2. Broker sends information to the client that the order was accepted for further processing.
3. Broker sends information to the client that 50 shares of given order were executed at price 779 and order is now partially filled.
4. Client requests the modification of limit price of given order from 780 to 790.
5. Broker sends information to the client that the modification request was received and the modification of order is in process.
6. Broker sends information to the client that the price of order was successfully modified to 790.
7. Broker sends information to the client that another 50 shares of given order were executed at price 789 and order is now fully filled.

2.6 Available FIX engines

Under the term FIX engine you can imagine a piece of software that manages a network connection, creates and parses outgoing and incoming messages, respectively, and recovers if something goes wrong. A FIX engine manages the session and application layers and is the single piece of software you need in order to FIX-enable trading or order management systems [FIXProtocolLtd09].

There exist a lot of FIX engines on the market and most of them can be found on the official FIX protocol website ([FIXProtocolLtd09]). With over a hundred engines listed any selection process would appear to be non-trivial.

All FIX engines do essentially the same thing but differ in three main ways:

1. **Capabilities/throughput** - which FIX versions are supported, how many messages per second can the engine process, does it support high availability and load balancing, what are the encryption options etc.
2. **Technologies used** - which platforms are supported, does it offer complete flexibility over your implementation through API or does it provide a ready-made functionality for many commonly used activities etc.
3. **Support and services offered** - what level of support is offered, is the vendor flexible on how you would like to pay, is source-code available and could it be modified etc.

In the following sections two engines, which are currently used for electronic trading in sponsoring company are very briefly described.

2.6.1 QuickFIX

QuickFIX ([QuickFix09]) is a full-featured open source FIX engine based on the Apache license that is compatible with the distribution of commercial software. Version 1.0 of QuickFIX was first released to the public in March 1st of 2002 and later that month was already adopted into several production systems. Current version is 1.12.4 and is widely spread in financial world. The QuickFIX core is implemented in C++, and comes with .NET, Java, Python and Ruby wrapper API's. It runs on all major OS systems, which include Windows, Linux, Solaris, FreeBSD and Mac OS X, and is currently compatible with the FIX 4.0-4.4 spec. QuickFIX is a simple and easy to use engine distributed in a form of library, which offers complete flexibility over your implementation.

2.6.2 CameronFIX Universal Server

The CameronFIX Universal Server ([Orc09]) is globally proven commercial FIX engine developed by Orc Software company - a premier member of the FIX Protocol Organization ([FIXProtocolLtd09]). It is a ready-made complex solution comprised of many modules providing advanced features like high availability, FAST (FIX Adapted for Streaming) support for reduction of bandwidth requirements and latency between sender and receiver, connection scheduling, direct market access through more than 100 available exchange adapters etc. Engine is very flexible and provides open architecture allowing the customers to customize the engine to their needs. It runs on any platform that supports Java including Windows, Solaris, HP/UX, AIX and LINUX and is compatible with all current versions (4.0 - 5.0) of FIX protocol.

As already mentioned at the beginning of this section, there are over a hundred engines. That fact should be taken into account during the proposal of the monitoring system. The system should not be tightly coupled to some specific version of FIX engine and should be able to monitor the trading infrastructure no matter of what engine is used for the underlying communication.

Chapter 3

Requirements analysis

3.1 Electronic trading infrastructure

Electronic trading infrastructure in sponsoring company is currently not much uniform. Some systems are purchased as a ready-made solution, some are being developed as a custom solution by external firms and some are developed internally. Each system provides some specific functions and they all together comprise a trading infrastructure necessary for the smooth running of business. As already mentioned in Chapter 2, many investment banks, exchanges and other market participants are using FIX as a messaging standard for electronic trading. Sponsoring company is no exception and FIX protocol is used for entire communication with clients and in a portion of communication between internal systems.

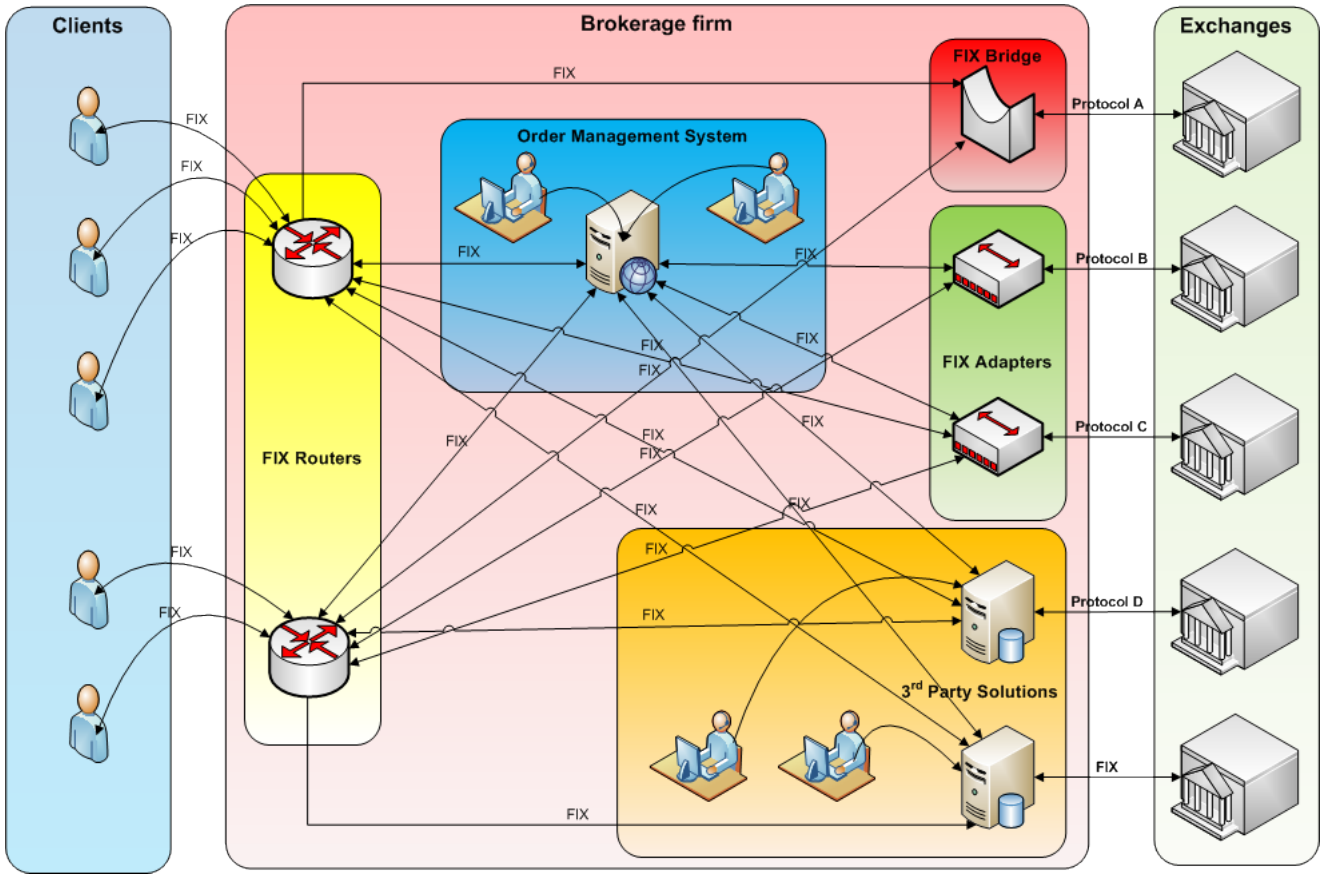


Figure 3.1: Current trading infrastructure

From Figure 3.1 you can see, that infrastructure is comprised of the following elements:

- **FIX router** - System for the correct routing of client messages to the target exchange or other alternative trading system. It is very often used as an entry point of trading infrastructure and ensures the communication between the institution and its clients. Correct routing is achieved by means of predefined rules.
- **FIX bridge** - System responsible for the translation of FIX messages into other systems specific protocol. It is frequently used as the last part of electronic trading chain and provides the communication between the institution and a stock exchange, where FIX protocol is not supported.
- **FIX adapter** - Provides the same functionality as FIX bridge with the difference that translation layer for exchange specific protocol is implemented as a plug-in. Therefore it is not necessary to implement FIX layer repeatedly and adapter can be used for different exchanges with different protocols at the same time.
- **OMS** - Order Management System is a central system for trading, used mainly by members of front office department, but may provide functionality for middle office and back office teams as well. Very often it is a fundamental system and some of them are so sophisticated that it is the only system of entire trading infrastructure. Its functionality may include:

  - A standards-based FIX API, which provides direct access to the major executing brokers and exchanges.
  - Dynamic routing based on all kinds of order parameters.
  - Flexible and powerful Order Entry interface, which ensures full control over an order.
  - Support for both manual and automated processing of orders.
  - Support for wide range of algorithmic order types providing traders with efficient executing strategies.
  - Support for real-time quotes and news feeds from the most popular and reliable market data vendors and news agencies.
  - Detailed logging for audit investigation purposes and trade disputes with customers.
  - Built-in reporting engine, which provides report formats for different departments of the company.
  - Wide range of predefined and customizable report formats that match the requirements of most popular clearing firms, banks and exchanges.

- **Third party solutions** - In the form of trading platform, which consists of backend server providing access to one or more markets and front-end trading terminal, which is used by traders. All of them supports a wide range of FIX protocols and interfaces to allow customers to communicate with system through FIX.

To simplify the whole trading infrastructure, the company is currently implementing number of propriatory algorithmic engines. It was designed as a distributed system, which will be used for both algorithmic trading and translation between FIX protocol and exchange specific protocols. Deployment of this system will result in replacement of older translation systems like FIX bridge and FIX adapters and in unification of trading infrastructure within the company. More details about the engine can be found in [Satanek09].

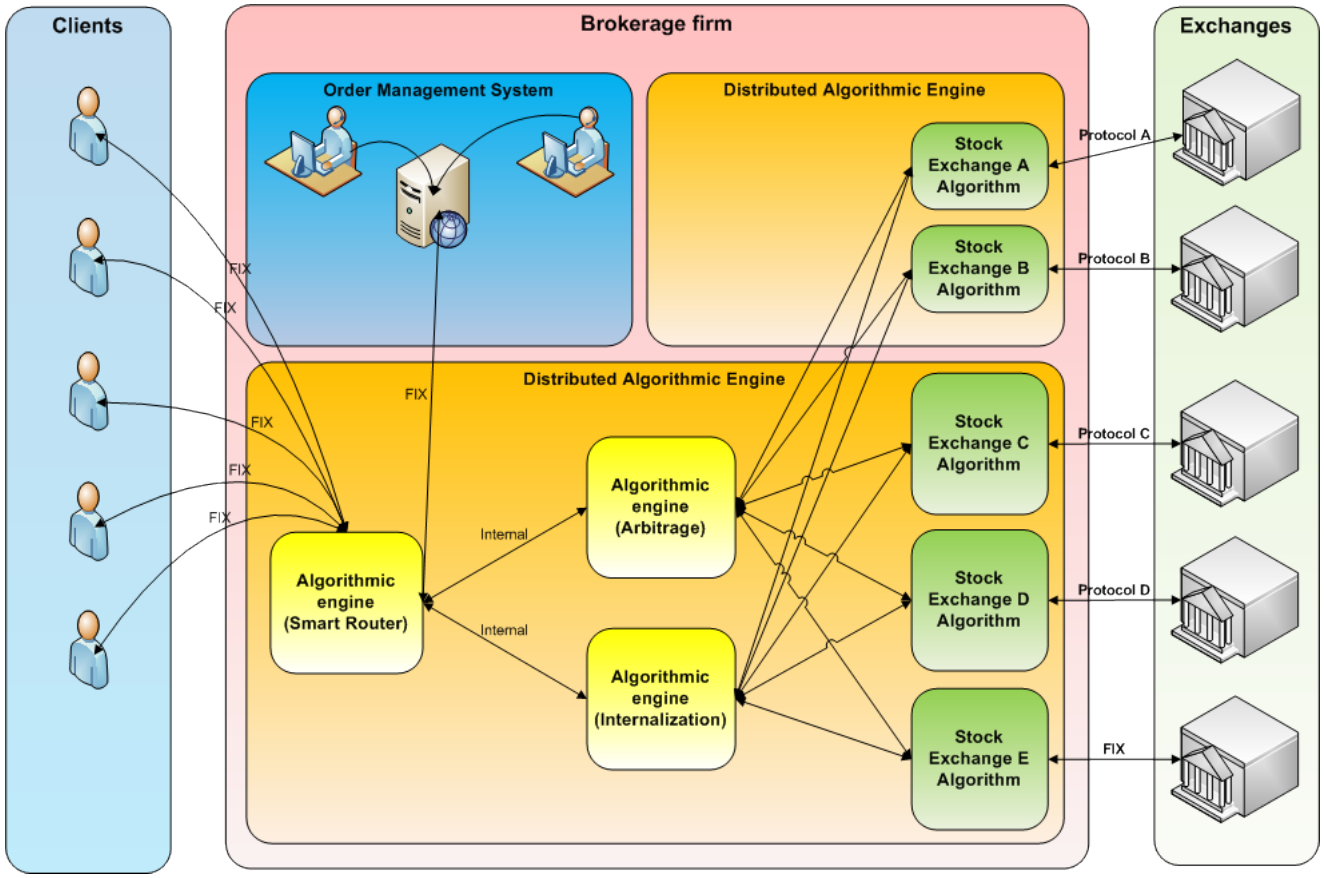


Figure 3.2: Trading infrastructure with distributed algorithmic engine

From Figure 3.2 you can see that the trading infrastructure with the use of algorithmic engines is much simpler. FIX bridge, FIX adapter and 3rd party solutions were removed and whole trading platform comprises from OMS for handling Care orders and a distributed algorithmic engine, which may include routing systems like smart order router, internalization engine or engine for arbitrage and systems with translation functionality for communication with individual stock exchanges. It is important to mention, that communication within individual components of distributed algorithmic engine is not through FIX protocol, but through internal protocol, which was designed specifically for this engine.

3.2 Current possibilities of monitoring

Sponsoring company currently does not dispose of monitoring system that would provide sufficient functionality to monitor the whole trading infrastructure. If we should summarize the main shortcomings than they are as follows:

- **Ability to monitor communication via FIX protocol only** - This will be insufficient for future infrastructure with distributed algorithmic engine, which will communicate through internal protocol.
- **Ability to monitor the input layer of trading infrastructure, this means FIX router systems only** - This allows monitoring communication between end clients and the brokerage firm, but not the flow of messages through other subsystems of the entire infrastructure.

- **Not designed for large data volumes** - System was created few years ago when message traffic was much lower than current one. It is able to smoothly monitor message traffic up to 30 thousand messages per day and higher traffic dramatically reduces user comfort.
- **Lack of filtering functions** - It is not possible to filter data according to all message properties and through the whole message hierarchy.
- **System is not modular** - Implementation changes must be done to add new markets and clients to the system.
- **Missing alerts support** - There is no alerting mechanism which lets the users know about possible problems during message processing.

From the above list of shortcomings is clear, that current monitoring tool is insufficient for efficient monitoring of company's trading infrastructure and new, feature rich system is needed.

3.3 Requirements for global monitoring system

Each monitoring system of electronic trading should provide its users the widest possible overview of the individual parts of the trading infrastructure and ensure the simplification of electronic trading environment. This should bring the organization reduced response times for messaging issues, and quick retrieval of information for reports and customer service requests. The following list is a specification of main and key features that a monitoring system should provide.

- **Current and future trading infrastructure support** - As already mentioned in Section 3.1, company is finishing the implementation of algorithmic engine, which should simplify the trading infrastructure. However, in practice deployment of new system to production environment is a matter of several months. The system must undergo extensive testing first and then, if all the tests pass, it is possible to start deployment of new system to production environment. None the less, there is no way to replace all obsolete solutions at once, as any overlooked bug in the system could lead to catastrophic consequences. Therefore, new systems are moved to production step by step, very often re-tested with end clients and it is necessary to count in the proposal with the presence of obsolete solutions, but also be prepared primarily for the future infrastructure comprised especially of distributed algorithmic engine.
- **Enterprise-wide infrastructure monitoring** - Enables tracking the progress of messages through internal systems and subcomponents. It is important for identifying problematic systems in trading chain in case of messaging issue.
- **High performance** - Designed for large data volumes. According to the current traffic and estimate of its future growth the system should be able to process message traffic up to millions messages per day across the whole trading infrastructure.
- **Comprehensive filtering functions** - Ability to define complex filter queries to search for a specific context in raw messages.
- **Error alerting** - Automatic checking for potential problems during the message processing and alerting the users.

- **Customizable order state overview** - Define custom and group-able views to display overview of number of orders in individual states. Allow user to immediately get the information about client's open orders on individual exchanges in case of connectivity problems.
- **P&L calculation** - Generate P&L reports for DMA traffic, so they can be viewed online or further processed by middle office team and combined with reports from order management system.
- **Customizable interface** - Flexible and functional design that will help users to create a personal interface tailored to their needs and tastes.
- **System configuration** - Easy setup of new users, markets and subsystems which should be monitored.
- **Reporting** - System to provide range of predefined report formats that match the requirements of most end clients. For those missing there should be option to define new formats in a form of plug-ins. It is necessary to be able to send reports according to a predefined time schedule in completely automatic mode.

3.4 Target platform

As sponsoring company has its own development department, it is important during the selection of target platform to take into account the requirements from the development department as well. This is the reason why the discussion on the selection of target platform is included already in this chapter. Most developers of sponsoring company specialize in the .NET framework development, specifically in C# language. Therefore a logical requirement of a monitoring system is its implementation in C# language, with which most of current developers of company has deep experiences, so they will be able to maintain and furthermore expand the system in the future. Though it may appear that C# is not appropriate language, mainly from performance point of view, it is not so. Financial institutions increasingly move their development to .NET framework and C# language, mainly due to the speed and comfort of the development opposed to competitive languages like Java and C++, but also because its performance is, in many cases, fully sufficient for trading applications. This illustrates, among other, Infolect system ([Microsoft06]), which is a trading platform used by the London Stock Exchange, one of the largest exchanges in the world.

3.5 Existing solutions

This section briefly describes some of available monitoring systems on the market and discusses their suitability for deployment at sponsoring company.

3.5.1 FIX Analyser

FIX Analyser is a FIX protocol monitoring tool developed by Onix Solutions [OnixS09], a financial services technology software business specialising in market data and trading access solutions. It is a very simple tool, which allows the user to load a list of FIX log files, detect changes in them and apply custom search filters. It supports different FIX log formats, so different FIX engines can be monitored at the same time, full text searches to find those messages which contain a text pattern, specific tag or specific text elements, customisable message highlighting, message details view and few other standard and useful functions.

The main problem of the tool is, that it is implemented as a thick client without server side, its user interface is not very friendly, doesn't support advanced features such as P&L calculation, reporting functions, ability to track the whole trading chain and many other fundamental features, which are required by a sponsoring company. The last stable and evaluated version of FIX Analyser is 1.8.0.23040.

3.5.2 FIX Eye

FIX Eye is a tool designed to facilitate fast data search in FIX log files and is developed by B2BITS company [B2BITS09], which delivers a wide range of solutions and consulting services for trading and operations in all asset classes. The functionality of FIX Eye product is similar to FIX Analyser mentioned in the previous section. It allows user to track changes of several FIX log files, parse the FIX messages and display them to the user in a readable format. It provides filtering functions to search messages by a particular text in fields or by a regular expression, supports all FIX dialects, custom views, message comparison etc. Even though its user interface is very intuitive and much more flexible than FIX Analyser's interface and it provides some additional functions, which are not present in FIX Analyser, such as basic alert notifications, order and session back-traces and few others, it still lacks same important features as FIX Analyser and therefore is also not suitable for deployment in sponsoring company. The last stable and evaluated version of FIX Eye is 2.1.10.14495.

3.5.3 MagniFIX

MagniFIX is an enterprise-wide FIX protocol monitoring tool developed by Greenline Financial Technologies [Greenline09], a provider of FIX and electronic trading technology solutions to optimize the trading environment. The MagniFix system is much more sophisticated solution than two previously mentioned tools, which were developed especially for a simple log analysis. It is developed as an enterprise-wide solution consisting of three different layers, a FIX collection layer, administration layer and visibility layer. FIX collection layer is represented by a set of agents responsible for collecting the FIX messages and passing them to an administration layer represented by an administration MagniFIX server. This server is the component which does the heavy lifting, searching, sorting, caching of FIX messages and even the alerting portion of thing is taken care of there. The visibility layer is then represented by a thin client, which displays all information to the end user and is connected to the administration layer, which provides the client with a requested data. The solution supports also a load balancing and distributed model, where a company can use several instances of MagniFIX administration servers to support heavier traffic loads. The other very nice feature of MagniFIX is its support for historical queries through which the user can retrieve messages from previous trading sessions. The MagniFIX also provides a very complex and flexible searching and filtering capabilities, same as a comprehensive alerting functionality. Shortcoming of whole MagniFIX solution lies in a missing support of trading chain tracking, P&L calculations and support to connect the system to external sources such as client database. It is also too much focused on a FIX protocol itself and therefore the communication between internal algorithmic engines can't be monitored at all.

Above introduced systems are not unique in the market, but not many others are available. Some of the worth mentioning ones - although they suffer from the same weaknesses as the above systems - are Daytona [FIXFlyer09] developed by FIX Flyer company [FIXFlyer09a], NetScout's [NetScout09] Sniffer Financial Intelligence [NetScout09a], which is focused especially on performance monitoring of FIX protocol-based applications, and TradeScope [NYFIX09], a monitoring solution developed by NYFIX Millennium LLC [NYFIX09a].

The conclusion is that as of today there are not many solutions on the market, which provide comprehensive monitoring features for whole trading infrastructure. There is no open source project on the market right now and the few commercial ones are not flexible enough and lack the basic requirements specified in the previous sections. All of them are too much focused on FIX protocol itself and its examination and do not provide important features like P&L calculation, reporting, ability to connect the system to external sources like client database, market data feeds etc. All systems also miss the ability to track messages through the whole trading chain in case of identifiers translations. All these shortcomings and the current offer on the market have led to a development of a custom project, which should meet all the requirements and provide comprehensive monitoring solution.

Chapter 4

System architecture

Following sections describe several different alternative architectures of system implementation, their main characteristics and suitability for monitoring system.

4.1 All-in-one client

Someone might think of the implementation as a standalone client where resides all the system functionality (see Figure 4.1). It is clear enough that all-in-one client without some centralised storage or service is not the right way of implementing such a monitoring system, because it would not be able to meet some major requirements specified in Section 3.3.

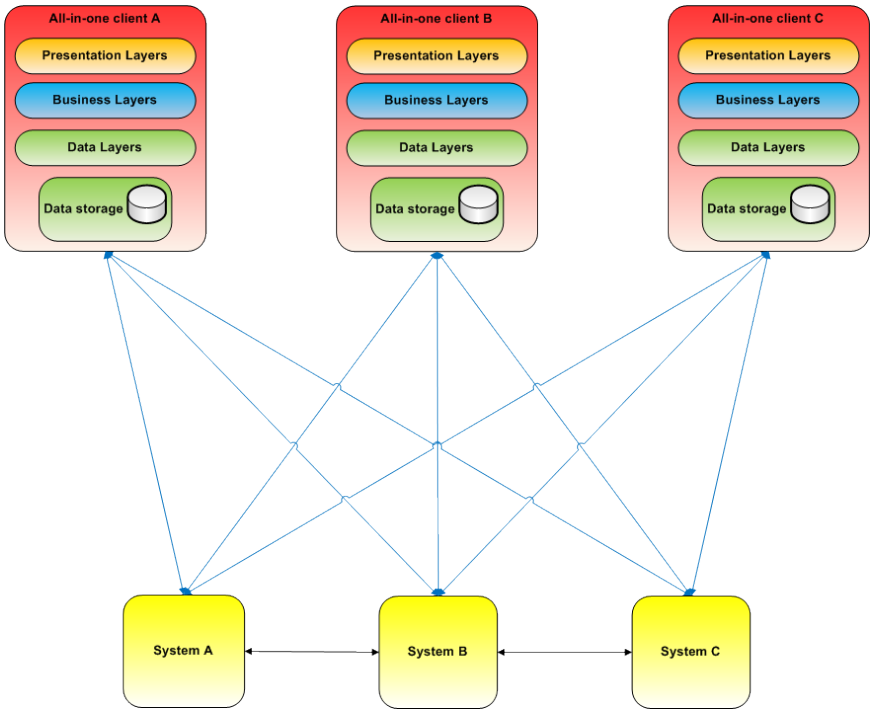


Figure 4.1: All-in-one client architecture

Characteristics of such a design are:

- **Ease of development and even deployment** - That makes this model very attractive, but for simple solutions only. At enterprise level application development it is used very rarely.
- **Client is responsible for everything** - It is responsible not just for a particular task, but can perform every step needed to complete a particular function.
- **High resource consuming** - Because all functionality resides on client, it may consume a lot of resources of workstation.
- **Not scalable** - To handle growing amounts of work in a graceful manner all computers with client instance have to be upgraded.
- **Difficult to maintain** - Because of need to recompile whole application with each repair or upgrade of particular code. Each upgrade then have to be deployed to all client workstations.
- **High hardware costs** - Because all data must be stored locally on a workstation, each workstation must be equipped with large disk capacity and processing power.
- **Large amount of network traffic** - Each client instance have to communicate with systems, which should be monitored.
- **Insufficient access control** - Because each client is independent from others and there is no centralised data storage, there is no sufficient mechanism to control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- **Impossibility to easily distribute data updates** - Without centralised data storage there is no way how to easily distribute common data updates to all running instances on the network and is both time-consuming and error-prone.
- **Sophisticated RDBMS systems can't be used** - Without centralised data storage it is not possible to take advantage of sophisticated RDBMS software, because each instance have to use its own data storage.

4.2 Two-tier client-server architecture

It is clear from previous section that one of the main disadvantages of all-in-one client implementation is a missing centralised storage. Therefore a two-tier client-server architecture model comes in mind. Client-server architecture describes the relationship between two computer programs in which one program, the client program, makes a service request to another, the server program. In turn, the server can accept the request, process it, and return the requested information to the client. The most basic type of client-server architecture employs only two types of hosts: clients and servers. This type of architecture is sometimes referred to as two-tier. It allows devices to share files and resources. The two-tier architecture means that the client acts as one tier and application in combination with server acts as another tier. There exist many types of servers such as web servers, ftp servers, name servers etc. In our implementation proposal the server will be represented by a database server, which provides a centralised data storage to all clients. You can see the monitoring system implemented as a two-tier client-server application in the following Figure 4.2.

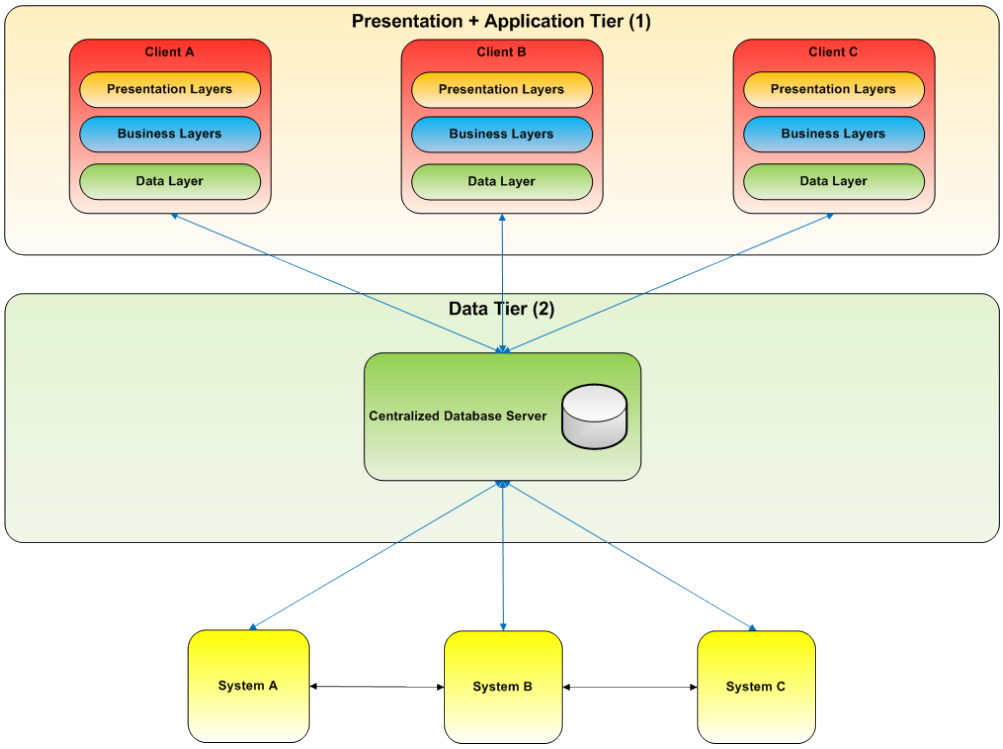


Figure 4.2: Two-tier client-server architecture

The main advantages against all-in-one client implementation are:

- **Central data management** - Since data storage is centralised, updates to that data are far easier to administer. This allows fast backups and efficient error management. Servers can better control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- **Higher performance and ability to process larger amount of data** - The server hardware is typically more powerful than desktop PCs and is designed to serve large amount of data.
- **Hardware costs can be minimized** - Because the data is not stored on each client, clients do not have to dedicate disk space and memory to storing data. The clients also do not need the processing capacity to manage data locally, and the server does not need to dedicate processing power to displaying data.
- **Reduced amount of network traffic** - All the data are processed on the server, and only the results are returned to the client. This reduces the network traffic between the client and monitored systems, improving network performance.
- **Easier maintenance and better scalability** - It is possible to replace, repair, upgrade, or even relocate a server while its clients remain both unaware and unaffected by that change.

Although the two-tier client-server architecture is significantly more suitable than the all-in-one client model, it still has some shortcomings, which include:

- **Still not very scalable** - The client application or server still have to run business logic. In both cases it is awkward. If you choose to embed the logic directly into the UI screens, then when the business logic got more complex, this code became very difficult to work with and furthermore it will easily lead to duplicate code, which means that simple changes resulted in hunting down similar code in many screens. On the other hand you can choose an alternative to put the business logic in the database as stored procedures. However, stored procedures give you limited structuring mechanisms, which again lead to awkward code [Fowler02].
- **Inefficient and slow data distribution** - Because all clients are connected directly to database, there is no efficient way how to distribute the data changes to the clients. It can be implemented on database level with triggers and some type of event notifications, but that is not suitable for high performance systems. The other technique when each client is requesting periodically data from database to reflect the changes is also inadmissible.
- **Insufficient concurrency checking, data validation and access control** - Without mediator between client and server, all concurrency checking, data validation and access control must be made on a database level with techniques like table locking, constraints and user privileges. This is not very efficient and difficult solution.
- **Database access layer still exposed to client** - Each client have to know the connection parameters to central database and have to access the database directly.
- **Expensive solution from RDBMS license point of view** - Many sophisticated RDBMS systems provide different licensing schemes. The two basic schemes, which are offered by almost all RDBMS vendors are: Per processor license: This is the simplest option, but also the most expensive. You pay a flat rate for each CPU(sometimes for each core) running RDBMS, and that's it. Server plus user CALs license: CAL stands for Client Access License. Under this scheme, you pay one price for the computer running RDBMS (no matter how many CPUs it has) and a separate price for each user that accesses the data. From these two schemes and two-tier client-server architecture characteristic it is clear that whether you choose a per processor license or server plus user CALs license, both will be expensive. Whether it's cheaper to go with per processor licensing or CALs depends on how many users your application will have. For internal applications it is mostly better to choose a scheme with CALs, of course if there are a limited number of users that will access the RDBMS. Let's show an example on a not named RDBMS system edition, which runs on a server with four processors and let's assume that monitoring system will be used by 60 users. In case of per processor license you have to pay $24,999 per processor, which gives you a total cost equal to $100,000. On the other hand if you go with a scheme with CALs it will cost you a $13,499 with 25 CALs included and $162 for each additional CAL. So the total cost is approximately $19,170. You can see that license scheme with CALs is more suitable for internal applications. But if you have a mediator, which ensures communication with the database, the pricing could be even better. In such a case you will need only a one CAL, so no additional costs for each user over the number 25 and you have further 24 free CALs, which can be used for other applications running on the same server.

As you can see, most of the above mentioned shortcomings are related to the fact, that all clients have to directly communicate with the database server. Therefore the last architecture - the multi-tier client-server one - is discussed in the following section.

4.3 Multi-tier client-server architecture

Multi-tier architecture (often referred to as n-tier architecture) is a client-server architecture in which the presentation, the application processing, and the data management are logically separate processes. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture [MultiTier09]. The main difference between two-tier architecture from monitoring system point of view is that the business logic can be pulled out from the presentation or data tier and, it can control an application's functionality by performing detailed processing from its own middle-tier. Business logic tier then can be implemented as a business logic server, which will host an API to expose business logic and business processes for use by thin clients and also by other applications (see Figure 4.3).

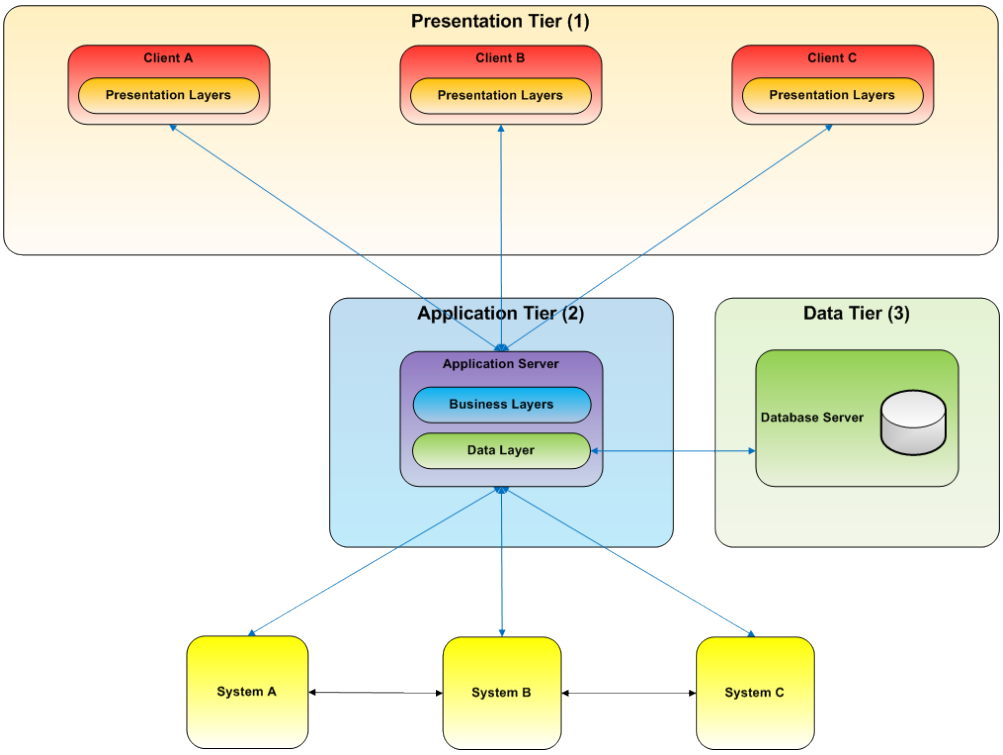


Figure 4.3: Multi-tier client-server architecture

Multi-tier architecture of monitoring system brings other advantages, such as:

- **Better concurrency checking, data validation and access control** - By centralising business logic to a separate process it is easier to check the concurrency, process data validation and control user permissions.
- **Ease of maintenance and better scalability** - Centralisation of the business logic holds clear advantages for maintenance and management. The computer that holds the business logic server can be maintained and upgraded as required to meet demand.
- **Thin client** - With this architecture much of the processing and data communications are moved to the business logic server and away from the workstation. Client is responsible for displaying information only.

- **Data Caching** - When the client workstation requests high volumes of data (such as viewing a large list of orders), data can be cached (or held) on the business logic server so that data is only transmitted as it is required. This means that the user can instantly view the first few orders without waiting for the entire list to be transmitted. It also reduces the impact of large data requests on the rest of the network.
- **Performance and fast data distribution** - Situating the business logic on a central computer decouples the clients from the processing. It brings the possibility to upgrade the business logic server on centralised computer in same way as a database server and to centralise better hardware on a single machine. With business logic server acting as a mediator of communication between client and database server, it is possible to distribute data to clients immediately when they are received from monitored systems and are pre-processed by the server.
- **Lower costs for RDBMS software and centralised data access** - Only one or few CALs are required for this type of architecture because database access layer is exposed only to the business logic server.
- **Better interoperability** - Centralised business logic server expose business logic to clients through several interfaces. This allows other applications to communicate with the server as well. At the same time business logic server can centralise access to other services or other tiers through service agents.

4.4 Conclusion

In above sections three possible architectures of monitoring system were discussed. During the discussion it became clear, that the first all-in-one client architecture is absolutely not suitable for such application like monitoring system. The main disadvantage is absence of centralised data storage, which is against many requirements like ability to process large amount of data, high performance, scalability and many others. The second two-tier client-server architecture appears to be a better solution because of centralised storage, but still has some disadvantages, which are not negligible. It suffers from inefficient and slow data distribution, insufficient concurrency checking and data validation and it is still not very scalable. As the most suitable from the monitoring system point of view appears to be a multi-tier client-server architecture. It provides the main advantages of other two architectures and has no significant disadvantage. Although it is not very easy to implement application with the multi-tier client-server architecture, it should bring many benefits in the long run and that should be kept in mind during the implementation.

Chapter 5

Middleware

In the previous chapter we have proposed that the most suitable architecture for monitoring system is multi-tier client-server architecture. Since this type of architecture forms a distributed system, an infrastructure that suitably supports the development and the execution of such distributed application is needed. A middleware platform presents such an infrastructure because it provides a buffer between the applications and the network. The network merely supplies a transport mechanism; access to it depends heavily on technological factors and differs between various physical platforms. Middleware homogenizes access to networks and offers generic services for applications. It also bridges technological domains and encapsulates the differences between different systems [Puder05].

The roles of middleware in distributed applications are many and can be logically grouped into communications middleware, database middleware, systems middleware and many others. In the following text we will discuss communications middleware, because it allows us to connect business logic server with thin clients and monitored systems and also to communicate with other services if needed.

5.1 Types of communication middleware and their suitability for monitoring system

An increasing variety of communication middleware styles is available for use in enterprises today. Within each of these styles, there are multiple products to choose from. Moreover, any of these products may be used alone or in combination with other products. Thus the problem of middleware selection is increasingly important in the engineering of enterprise software systems [Sutton00]. A complete, formal analysis of communication middleware would be a daunting task. The list of options and functionality of middleware is incredibly long, particularly when one considers, for example privacy, non-repudiation, transactions, reliability, and message sequence preservation. Therefore only the most popular communication middleware styles are introduced in this section and their suitability is discussed especially from the monitoring system architecture point of view. The main requirements for such middleware are as follows:

- **Time and synchronization decoupling** - The interacting parties do not need to be actively participating in the interaction at the same time. In particular, the sender might send some messages while the receiver is disconnected, and conversely, the receiver might get notified about the occurrence of some message while the original sender of the event is disconnected. Senders are not blocked while sending messages.

The production and consumption of messages do not happen in the main flow of control of the senders and receivers, and do not therefore happen in a synchronous manner. This fact is very important for both communication between monitored systems and business logic server and communication between business logic server and thin clients. Because trading systems are standalone and crucial applications in trading infrastructure, their operation must not be affected or slowed down in case of business server failure. Someone would suggest that this situation can be solved with sophisticated exception handling for transaction failures followed by data retransmission when the communication between endpoints is up again, but this behaviour is not desirable because of high frequency data processing and large amount of data buffered on side of trading system in such situation. The same applies for communication failure between business logic server and thin client.

- **Space decoupling** - The interacting parties do not need to know each other. The sending party is sending messages and the receiver get these messages indirectly. The senders do not usually hold references to the receivers, neither do they know how many of these receivers are participating in the interaction. Similarly, receivers do not usually hold references to the senders, neither do they know how many of these senders are participating in the interaction. This type of decoupling is important because of data distribution from business server to thin clients. Business server does not need to know how many thin clients are connected and where they are running.
- **Performance of data distribution** - Ability to transfer message from sender to many receivers in the shortest possible time. Ability to transfer high rates of messages per second to many receivers. Keep network traffic load as low as possible.
- **Platform and language independence** - It is important to have an ability to deploy client and server on different platforms and to have a free choice, in which language particular parts of system will be implemented.
- **Interoperability** - Different implementations from different vendors should be compatible, so one is not dependent on same vendor for maintenance support and future enhancements.

5.1.1 Remote Procedure Call

Remote Procedure Call (RPC) is a client/server infrastructure that increases the interoperability, portability, and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms [RPC09]. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces - function calls are the programmer's interface when using RPC [Rao95]. This type of communication has first been proposed in 1983 by Andrew D. Birrell and Bruce Jay Nelson in the form of Remote Procedure Call (RPC) for procedural languages.

- **Time and synchronization coupling** - RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. The use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. This leads to strong time and synchronization (on the consumer side) coupling.

- **Space decoupling** - Since an invoking object holds a remote reference to each of its invokers it introduces a strong space coupling.
- **Interoperability** - RPC implementations are nominally incompatible with other RPC implementations, although some are compatible. Using a single implementation of a RPC in a system will most likely result in a dependence on the RPC vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability [SEI08].
- **Platform and language independence** - Because RPC is a procedural mechanism, it is not very common that it supports more languages. Platform independence is on the other hand a common scenario, where RPC is used.
- **Performance of data distribution** - One of the strengths of RPC is that the synchronous, blocking mechanism of RPC guards against overloading a network, unlike the asynchronous mechanism [Rolstadas00]. However, when recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network [SEI08]. The other fact is, that since an invoking object holds a remote reference to each of its invokers, the data have to be distributed from one publisher to several receivers separately, which could lead to higher traffic load and slow data distribution as well.

It is clear, that RPC does not meet any of the requirements of monitoring system and the technique of procedural programming is little bit outdated. However the main reason, why the RPC was briefly described in previous paragraph is that many other middlewares, including the newest ones, use its model for communication and enriches it by some other features.

5.1.2 Object Request Broker

RPC applied to object-oriented contexts are often called Object Request Broker (ORB). The main difference between RPC and ORB is that whereas the ORB is object-oriented in nature, RPC is procedural. That means, that a logical pipe exists between two objects, rather than two procedures, which are physically apart on two different networks and interacting via this pipe. ORBs promote interoperability of distributed object systems because they enable users to build systems by piecing together objects from different vendors that communicate with each other via the ORB. This type of communication results in many models for distributed computation such as remote method invocations in Java RMI [Sun00], CORBA [OMG08], .NET Remoting [Microsoft09] and many others. In the following paragraphs we will discuss most popular ORB-like models with support of .NET framework only, so e.g. Java RMI is not included in our proposal.

5.1.2.1 .NET Remoting

.NET Remoting [Microsoft09] is a Microsoft application programming interface (API) for interprocess communication released in 2002 as a part of the 1.0 version of .NET Framework. It enables developer to build widely distributed applications easily, whether application components are all on one computer or spread out across the entire world. One can build client applications that use objects in other processes on the same computer or on any other computer that is reachable over its network [Microsoft09a]. One of the advantages of .NET Remoting is, that one can use different transport protocols, serialization formats, object lifetime schemes, and modes of object creation.

Figure 5.1 shows the general remoting process.

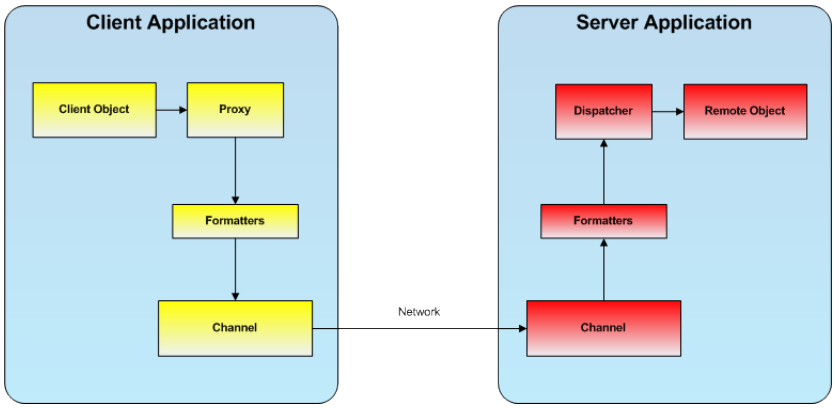


Figure 5.1: .NET Remoting communication process

There are several key players within .NET Remoting communication process. Client Object is the object or component that needs to communicate with a Remote Object. When the Client Object needs to call a method from the Remote Object it uses a Proxy object to do this. Every public method that is defined in the remote object class can be made available in the proxy and thus can be called from clients. The proxy object acts as a representative of the remote object. It ensures that all calls made on the proxy are forwarded to the correct remote object instance. Messages are transported through channels. You can choose a TcpChannel or an HttpChannel or extend one of these to suit your requirements. Each message is before and after transmission formatted by formatters on each side of the communication. Once the message arrives to the server process a listening channel forwards the request to the dispatcher, which locates and calls the requested object. The process is then reversed. The server system bundles the response into a message that the server channel sends to the client channel. Finally, the client system returns the result of the call to the client object through the proxy.

If we look at how .NET Remoting meets the requirements of monitoring system, we realize that the biggest disadvantage is, that .NET Remoting is not interoperable with other systems. The dependence on assembly metadata implies that client applications must understand .NET concepts, which results in tight coupling to Windows platform and .NET framework. Other disadvantage of .NET Remoting is that it is strongly time coupled and there is no built-in support for standard data distribution mechanisms, such as publish-subscribe through multicast protocol, which leads to same performance issues as in RPC. More details about technology can be found at [Microsoft09a] and in [Microsoft09].

5.1.2.2 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) is a specification of a standard architecture for object request brokers. It was defined by the Object Management Group (OMG) [OMG09]. The OMG was established in 1988, and the initial CORBA specification emerged in 1992. Since then, the CORBA specification has undergone significant revision, with the latest revision (CORBA v3.1) released in January 2008 [OMG08].

Figure 5.2 illustrates the high-level paradigm for remote interprocess communications using CORBA.

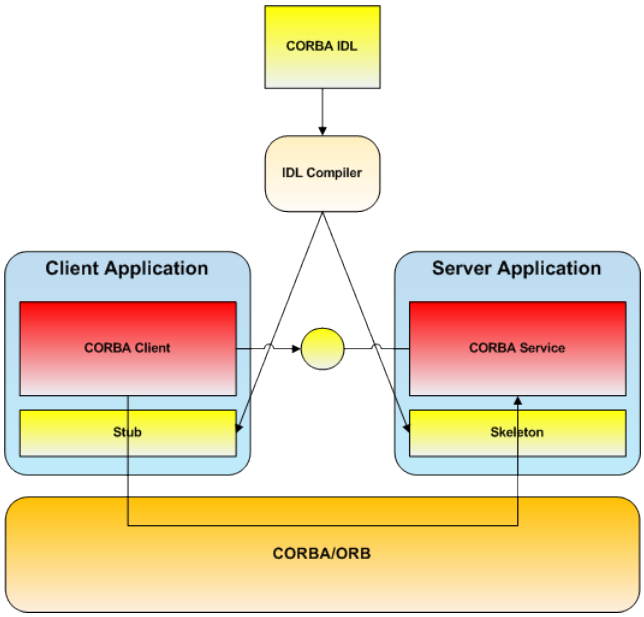


Figure 5.2: CORBA remote interprocess communication

CORBA uses an interface definition language (IDL) to specify the interfaces that objects will present to the outside world. CORBA then specifies a "mapping" from IDL to a specific implementation language. Standard mappings exist for C, C++, Java and many other languages. When you have interfaces described by IDL, language specific IDL compiler can be used to generate an IDL stubs and skeletons for each described object. Stub is a client-side definition and skeleton a server-side. The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object. The ORB locates the appropriate implementation code, transmits parameters, and transfers control to the Object Implementation through an IDL skeleton. When the request is complete, control and output values are returned to the client [OMG08]. CORBA specifies the General Inter-ORB Protocol (GIOP), an abstract protocol by which ORBs communicate. The GIOP architecture provides several concrete protocols and the most important one is The Internet Inter-ORB Protocol (IIOP) for communication over TCP/IP layer.

The main advantages of CORBA against previously described technologies are:

- **Interoperability** - Since the CORBA is a standard specification, different implementations from different vendors could be compatible in terms of basic functionality at least.
- **Language independence** - Client and server can be developed independently and in different programming languages. Interface definition language used by both client and server establishes the interface contract between them and is the only thing they need to agree on. Many mappings between IDL and programming languages exist.
- **High performance data distribution** - By using the Event and Notification Services of CORBA implementation, one can use multicasting of messages from publisher to many subscribers in an asynchronous manner, which increases data distribution performance. Moreover structured and filtered events allow more efficient processing of the event notification and cut down on traffic load.

- **Many other useful features** - During evolving of CORBA specification, many additional CORBA services were defined. Each service brings a new set of features to CORBA, which can be used to different communication scenarios. Some of these services are Persistent State Service, which provides a way of making a CORBA service persistent, Event Service and Notification service, which defines an event data delivery model that allows decoupled communication between suppliers and consumers of events, CORBA Messaging Service for Asynchronous Method Invocation (call-back model), Quality of Service (QoS) messaging and many others.

There are many CORBA implementations on the market today. Some of them are commercial and many are published as an open source products. The most popular open source CORBA implementation today is a TAO (The ACE ORB) [Wustl07]. TAO is developed at Washington University and introduces advanced, CORBA V3.0 compliant, C++, real-time Object Request Broker (ORB). It is designed to meet the stringent Quality of Service (QoS) requirements of real-time applications, resulting in superior end-to-end predictability, efficiency, and scalable performance [OCI09]. It implements many of the OMG CORBA services specifications and is implemented on top of the ADAPTIVE Communication Environment (ACE) [Wustl07a], which is the lower level middleware that implements the core concurrency and distribution patterns for communication software. There are several companies, which support ACE and TAO distribution commercially such as PrismTech [PrismTech09] or OCI [OCI09].

The main disadvantage of CORBA specification, from the monitoring system point of view, is that although there exist mappings of IDL to many different programming languages, it has completely ignored .NET Framework and its main languages such as C# and Visual Basic. Despite this, an open source project IIOP.NET [IIOP.NET04] allows an interoperation between .NET and CORBA by incorporating CORBA/IIOP support into .NET, leveraging the .NET Remoting framework by providing a custom channel that supports the IIOP protocol. IIOP.NET project is maintained by ELCA Informatique SA [ELCA09] and was developed in collaboration with the Programming Languages and Runtime Systems Research Group of the ETH-Zurich [ETH03] as part of Dominic Ullmann's diploma thesis. The project is licensed under LGPL license, so commercial usage is not an issue.

The other very important technical problem is CORBA's complexity. It is not very easy to start development with CORBA from beginner's perspective. It takes a long time to read all the CORBA specifications, learn the IDL, understand how things work internally etc. It might make sense to use CORBA in an environment with pre-existing CORBA infrastructure but, otherwise, it seems there is not much going on anymore. And starting a new development with CORBA now would seem a rather risky investment, seeing that there is only a few commercial vendors left, and that activity around the open source versions seems to be diminishing too [Henning06]. More details about CORBA specification can be found at [OMG09] and in [OMG08].

5.1.2.3 Internet Communications Engine

The Internet Communications Engine (Ice) is a modern object-oriented middleware developed by ZeroC [ZeroC09]. Ice is free software, available with full source, and released under the terms of GNU General Public License (GPL). Commercial licenses are available for customers who wish to use Ice for closed-source software [ZeroC09].

In concept Ice is similar to CORBA, and indeed was created by several influential CORBA developers, including Michi Henning, who worked on Corba as an ORB implementer, consultant, and trainer. In some areas, Ice is remarkably close to CORBA whereas, in others, the differences are profound and have far

reaching architectural implications [Henning09]. Infrastructure is illustrated in Figure 5.3, which is taken from [Henning09].

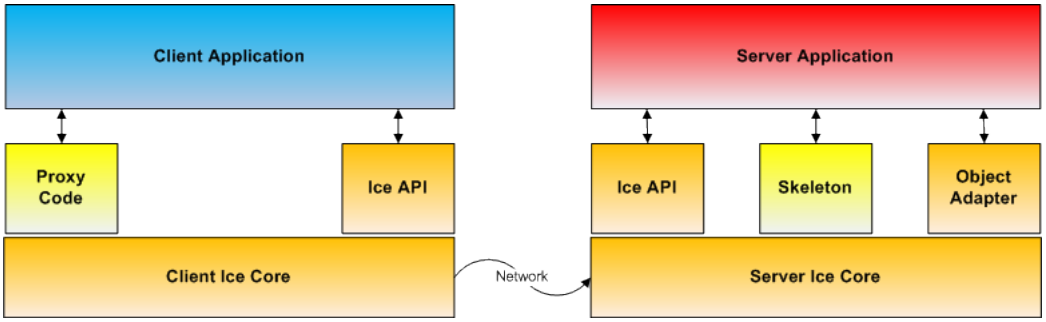


Figure 5.3: Ice client and server structure

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions. Slice (Specification Language for Ice) allows developer to define the client-server contract in a way that is independent of a specific programming. These definitions are compiled by a Slice compiler into an API for a specific programming language. The client side generated code is called Proxy and a server side is called Skeleton. Ice Core is a run-time support for remote communication and much of its code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that should be kept away from application code. Proxy is responsible for sending an RPC message to the server that invokes a corresponding function on the target object and for marshalling and unmarshalling the code. The server-side equivalent Skeleton provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The last important component is the object adapter, which maps incoming requests from clients to specific methods on programming-language objects, is responsible for the creation of proxies that can be passed to clients and is associated with one or more transport endpoints [Henning09]. So if you compare the design with CORBA specification, it is clear that Ice is very close to it. In spite of that there are some important differences between them and will be mentioned in the following paragraphs.

Same as CORBA specification defines some optional services, ice ships with a number of services that provide features enriching the base remoting capability of protocol. The services are implemented as Ice servers to which your application acts as a client. Descriptions of services are taken from [Henning09].

- **Freeze** - Built-in object persistence service. Freeze makes it easy to store object state in a database. Developer define the state stored by his objects in Slice, and the Freeze compiler generates code that stores and retrieves object state to and from a database.
- **IceStorm** - Object-oriented publish-subscribe framework that also supports federation and quality-of-service.
- **IceGrid** - Implementation of an Ice location service that resolves the symbolic information in an indirect proxy to a protocol–address pair for indirect binding. It also supports replication, load-balancing and failover.
- **IceBox** - Simple application server that can orchestrate the starting and stopping of a number of application components.

- **IcePatch** - Facilitates the deployment of ICE based software. For example, a user who wishes to deploy new functionality or patches to several servers may use this service.
- **Glacier** - Ice firewall traversal service. It allows clients and servers to securely communicate through a firewall without compromising security.

All these services allow developer to focus on application development instead of having to build a lot of complex infrastructure first.

If we compare the Ice against CORBA specification, the following points describe the most important advantages of Ice:

- **Complexity** - CORBA is known as a platform that is large and complex. This is largely a result of the way CORBA is standardized: decisions are reached by consensus and majority vote. In practice, this means that, when a new technology is being standardized, the only way to reach agreement is to accommodate the pet features of all interested parties. It results into specifications that are large, complex, and burdened with redundant or useless features. In turn, all this complexity leads to implementations that are large and inefficient. The complexity of the specifications is reflected in the complexity of the CORBA APIs: even experts with years of experience still need to work with a reference manual close at hand, and, due to this complexity, applications are frequently plagued with latent bugs that do not show up until after deployment. In contrast to CORBA, Ice is a simple platform. Feature set is both sufficient and minimal, is easy to learn and understand the platform, and it leads to shorter development time with lower defect counts in deployed applications [Henning09].
- **Language support** - Supports all most popular languages such as C++, Java, Python, Ruby, PHP and finally C# (and other .NET languages, such as Visual Basic). which was ignored in CORBA specification.
- **Wider range of built-in features** - As already mentioned, CORBA platform provides specifications of many services to enrich the basic RPC communication mechanism. The problem is, that most of them are either optional or not widely implemented so, as a developer, you are typically faced with having to choose which feature to do without. Typically it is impossible to find all features in a single implementation. On the other hand, Ice brings wide range of services built into the platform, which cover mostly used optional services in CORBA. At the same time, Ice introduces some other techniques, for which there is no equivalent in CORBA specification. These are Asynchronous Method Dispatch (AMD) used to suspend processing of an operation in the server, services allowing coexistence with firewalls, Ice protocol supporting the UDP (both unicast and multicast) as well as TCP and many others.
- **Very well documented** - Ice documentation includes in-depth description of all features, many code examples and real-world scenarios. Developer community is also very active and provides support in developer forums.

More detailed discussion about advantages of Ice over CORBA can be found in [Henning04] and [Henning06].

Ice is used in many mission-critical projects by companies all over the world ranging from battlefield simulations to real-time stock trading systems and the only disadvantage, which could someone point out is already mentioned dual licensing. So in case one wants to use the middleware in commercial sphere, it can't be used free of charge and proprietary license have to be bought.

5.1.3 Data Distribution Service

Data Distribution Service for Real-time Systems (DDS) [OMG07] is an open standard of a Data-Centric Publish-Subscribe (DCPS) programming model for distributed systems. A few proprietary DDS solutions have been available for several years, until 2004 when the two major DDS vendors, the American Real-Time Innovations [RTI09] and the French Thales Group [TG09] teamed up to create the DDS specifications that have been approved by the Object Management Group [OMG09] [DDS09]. As of June 2009, the latest official version of standard is DDS 1.2 released in January 2007.

Specification of DDS describes two levels of interfaces [OMG07]:

1. A lower DCPS (Data-Centric Publish-Subscribe) level that is targeted towards the efficient delivery of the proper information to the proper recipients.
2. An optional higher DLRL (Data Local Reconstruction Layer) level, which allows for a simple integration of the Service into the application layer.

Data-centric publish-subscribe model was designed to be high-performance and predictable as well as efficient in its use of resources. It builds on the concept of a 'global data space' that is accessible to all interested applications. Applications that want to contribute information to this data space declare their intent to become Publishers. Similarly, applications that want to access portions of this data space declare their intent to become Subscribers. Each time a Publisher posts new data into this global data space, the middleware propagates the information to all interested Subscribers [OMG07]. In Figure 5.4 is outlined the basic conceptual model of DDS.

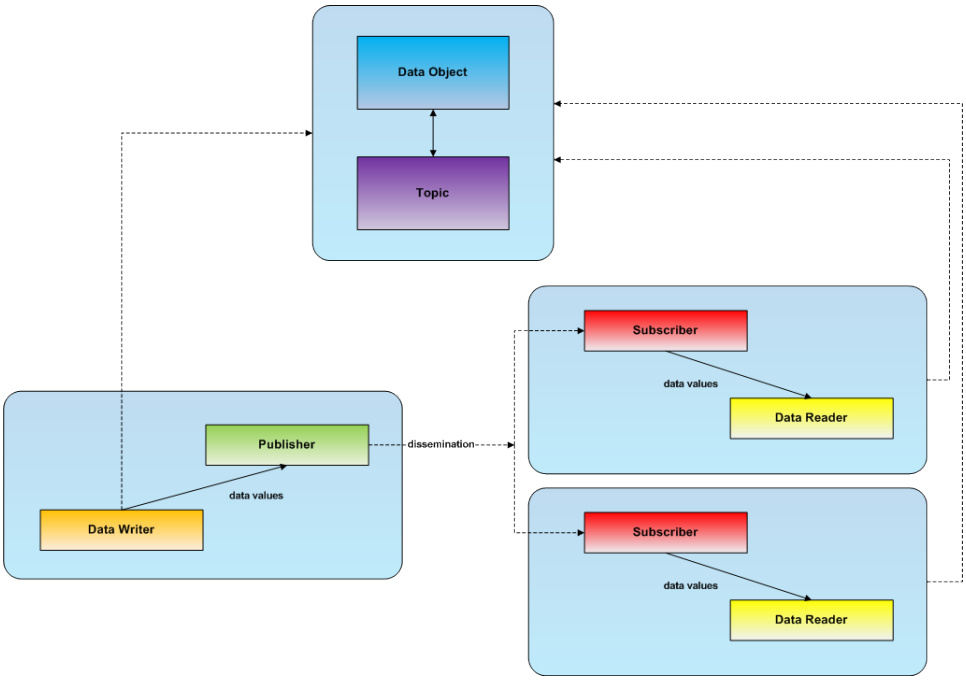


Figure 5.4: Data Distribution Service model architecture

In addition to publishers and subscribers, which are responsible for data distribution and for receiving published data, some other objects are present. The DataWriter is the object the application must use to

communicate to a publisher the existence and value of data-objects of a given type. On the other hand to access the received data, the application must use a typed DataReader attached to the subscriber. Topic objects conceptually fit between publications and subscription. It associates a name, a data-type, and QoS related to the data itself.

The behaviour of the service is controled by QoS policies. Each entity of model supports its own specialized kind of QoS policies. Topic QoS, the QoS of the DataWriter associated with that Topic and the QoS of the Publisher associated to the DataWriter control the behaviour on the publisher's side, while the corresponding Topic, DataReader, and Subscriber QoS control the behaviour on the subscriber's side. There are many QoS policies defined through which you can configure different aspects of data availability, delivery and timeliness. QoS can be used also to control and optimize network as well as computing resources [OMG07].

As of today there are couple of open source implementations of DDS on the market and several proprietary products. The most comprehensive open source implementation is a PrismTech's OpenSplice [PrismTech09], a 2nd generation fully compliant OMG-DDS implementation. PrismTech Corporation [PrismTech09a] offers their DDS technology organized around four different editions, from which the community edition is available as open source and is licensed under LGPL and other value-add editions are accessible via commercial subscriptions. The community edition supports the full implementation of DDS specification except the optional Data Local Reconstruction Layer, which is part of commercial editions only. It supports Linux and Windows as operation systems and provides language API for C, C++ and Java. Unfortunately as of today, C# language support is not available, but is announced to be released in few months as a part of community edition too.

The focus on high-performance data distribution allowing ultra low latencies and ultra high throughput, support for many QoS policies allowing high availability, load balancing, and a fully decoupled interoperable architecture supported on many platforms with main language APIs makes from a Data Distribution Service definitely a very interesting technology. It is already deployed in very demanding mission and business-critical systems, ranging from automated financial trading to air traffic management. From the monitoring system point of view, the technology is very promising and the only disadvantage is missing C# language API, which should be eliminated in couple of months. The other fact is that technology might be a little bit complex, so the learning curve could potentially be longer than with other technologies.

5.1.4 Message-Oriented Middleware

Message-oriented middleware (MOM) is a client/server infrastructure that increases the interoperability, portability, and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It hides from the developer the details about different operating systems and network protocols, so he can concentrate on application development itself. Programming Interfaces (APIs) that extend across diverse platforms and networks are typically provided by the MOM [Rao95].

The main advantage of a message-based communications protocol lies in its ability to store, route or transform messages in the process of delivery. Most MOM systems provide persistent storage to back up the message transfer medium. This brings ability to handle delays between a request and a response and the mechanism is very useful when dealing with disconnections, such as unreliable networks or unreliable receiver application. When such an application fail for any reason, the senders can continue unaffected, as the messages they send will simply accumulate in the message store for later processing when the receiver restarts.

Routing ability of MOM middleware layer itself allows to route messages differently for each communication scenario. Many MOM products offer routing mechanisms such as broadcasting and multicasting for fast data distribution from one sender to many receivers. Built-in intelligence of MOM system can transform messages to match the requirements of the sender or of the recipient. In conjunction with the routing and broadcast/multicast facilities, one application can send a message in its own native format, and two or more other applications may each receive a copy of the message in their own native format.

These characteristics make a message-oriented middleware a good candidate for monitoring system implementation because of its asynchronous behaviour and time-decoupling allowed by storage, space-decoupling and possible fast data distribution allowed by routing mechanisms and interoperability because of message transformation.

The primary disadvantage of message oriented middleware systems is its requirement of an extra component in the architecture, the message transfer agent. As with any system, adding another component can lead to reductions in performance and reliability, and can also make the system as a whole more difficult and expensive to maintain.

5.1.4.1 Java Message Service

Most MOM vendors provide solutions based on a Java Message Service (JMS) API [Sun02], which is a standard API introduced as a part of the Java Enterprise Edition (JEE) platform. Figure 5.5 illustrates the architecture of JMS API programming model.

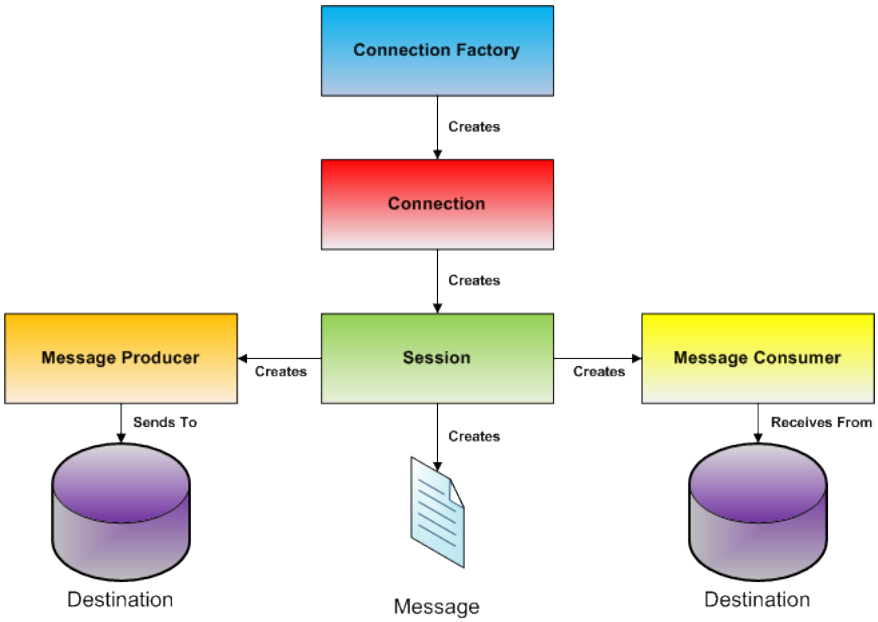


Figure 5.5: Java Messaging Service API programming model

A connection factory is the object a client uses to create a connection with a JMS provider. Connection then encapsulates a virtual connection with a JMS provider and represents an open TCP/IP socket between a client and a provider service daemon. Each connection is responsible for creating sessions, which have three tasks: creating message producers, that will be sending messages to target destination, creating message

consumers interested in message consumption and finally creating messages, which will be distributed between producers and consumers through particular destinations.

JMS defines two basic types of messaging domains: point-to-point model and publish-subscribe model.

- **Point-to-point model** - Is built around the concept of message queues. Sender sends a message to a specific queue, which retain this message (and each consequtive one) until it is consumed by the receiver or until it expires. Within this model each message has only one consumer. Point-to-point model is illustrated in Figure 5.6.

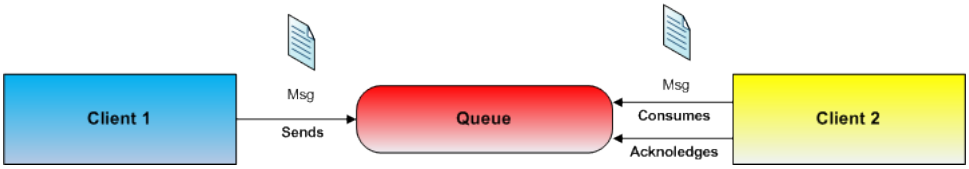


Figure 5.6: Point-to-point messaging domain

- **Publish-subscribe model** - Is built around the concept of topics. Counterparty, which is addressing the messages to a topic is called publisher and receivers of messages are called subscribers. They are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The messaging system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. This model allows distribution of one message to many receivers. Topics are commonly used without the need to retain messages and that is also the default behaviour. But if a client wants to receive a message published during the time the client was not subscribed to the topic, the client can use a durable subscription, which provides the flexibility and reliability of queues but still allow clients to send messages to many recipients [Sun09]. Publish-subscribe model is illustrated in Figure 5.7.

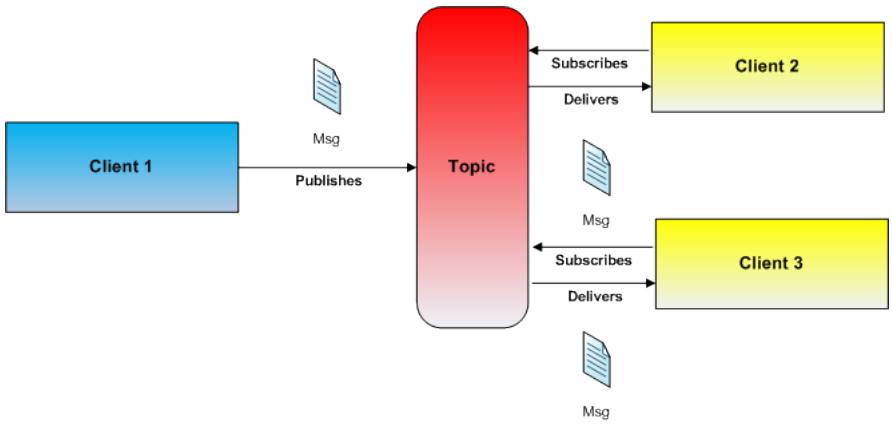


Figure 5.7: Publish-subscribe messaging domain

Other interesting JMS feature, which is provided by many MOM vendors are message selectors. Selectors allow a message consumer to specify the messages it is interested in. This interest is defined with a SQL-like syntax, which is applied against the message properties and filtering takes place on the JMS provider itself.

There are many free, open source and proprietary providers of JMS-based middleware on the market today. Most of them, as already mentioned, supports the functionality defined in JMS, but differs with vendor specific advanced features like failover support, clustering, wildcard-based filtering of messages, special queue types support, different performance skills and many others. What should be mentioned here is, that since JMS does not define the format of the messages that are exchanged, JMS systems from different vendors are not interoperable. All the major vendors have their own implementations and if you choose a product from one vendor, it can't be replaced by other product from different vendor in the future.

5.1.4.2 Advanced Message Queuing Protocol

Advanced Message Queuing Protocol (AMQP) [AMQPWG09] originated in the financial services and was developed from 2004 to 2006 by JPMorgan Chase & Co. [JPMorgan09] and iMatix Corporation [iMatix09]. JPMorgan Chase & Co. and iMatix documented the protocol as an interoperable specification and assigned it to a working group, which as of June 2009 has eighteen members. Because the protocol was developed at first especially for financial sector, many members of working group are also from this sector, eg. Barclays Bank PLC, Credit Suisse, Deutsche Börse Systems, Goldman Sachs, Tervela Inc. and already mentioned JPMorgan Chase & Co. All of these banks are major leaders in finance world. But not only finance industry is involved, companies like Microsoft Corporation, Cisco Systems, Red Hat and Novell are interested also and many others.

AMQP is the industry's first standard for messaging that spans from the wire-level to the semantics of messaging. This is significant because before the arrival of AMQP, no two messaging implementations could natively interoperate with each other—even though messaging software's core mission is to distribute data across disparate systems. Such a specification is, of course, very useful for a lot of customers - probably in a similar way to how the standardisation on TCP/IP has made networking so much easier [RedHat08].

A general architecture of an AMQP compliant messaging system is illustrated in Figure 5.8.

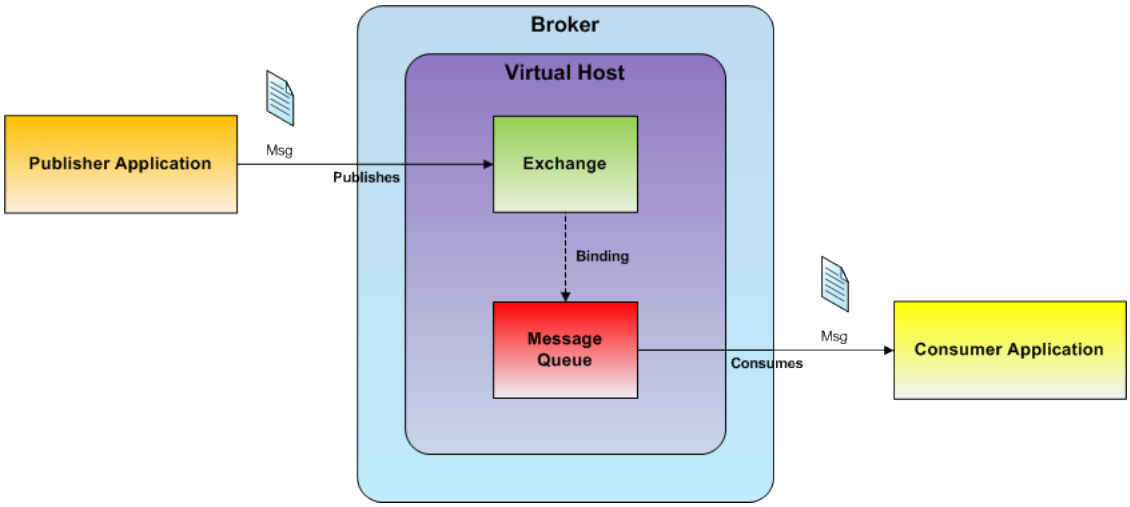


Figure 5.8: Advanced Message Queuing Protocol architecture

An AMQP messaging system consists of three main components: Publisher(s), Consumer(s) and Broker(s). Each component can have multiple instances and these instances can be situated on independent hosts. Publishers and Consumers communicate with each other through message queues bound to exchanges within

the Brokers. AMQP provides reliable, guaranteed, in-order message delivery. In the following paragraph is briefly explained the functionality of each component. Descriptions are taken from [AMQPWG09a]

- **Publisher** - It is an application that constructs messages with AMQP compliant headers.
- **Consumer** - An application that receives the messages from one or more Publishers.
- **Message Queue** - Stores messages in memory or on disk drive, and delivers these in sequence to one or more consumer applications. Message queues are message storage and distribution entities. Each message queue is entirely independent.
- **Exchange** - Accepts messages from a producer application and routes them to message queues according to prearranged criteria. These criteria are called Bindings. Exchanges are matching and routing engines. That is, they inspect messages and using their binding tables, decide how to forward these messages to message queues. Exchanges never store messages.
- **Binding** - A relationship between a message queue and an exchange. The binding specifies routing arguments that tell the exchange which messages the queue should get. Applications create and destroy bindings as needed to drive the flow of messages into their message queues. The lifespan of bindings depend on the message queues and exchanges they are defined for - when a message queue, or an exchange, is destroyed, its bindings are also destroyed.

Each message has a header consisting of a defined set of properties, and a body that is an opaque block of binary data. Each message header contains a text string known as the Routing Key. The Routing Key is a virtual address that the exchange may use to decide how to route the message.

AMQP defines many other features like different types of Exchanges for different communication models, different types of QoS, transactions etc. All these functionality introduced as open standard brings to the world of messaging a new light. If we summarize the AMQP from the monitoring system point of view, the protocol meets all the requirements because it is a standard of message-oriented middleware, its advantages were discussed in previous sections, which brings the missing interoperability in the world of messaging. The other fact, which should be noted is that since the protocol has a strong base in financial sector, it is tailored for the levels of performance required by the financial services industry, so the performance should not be issue too. The protocol was published till now in four versions, the last one is AMQP/0-10, which is closest to the forthcoming AMQP/1.0. There are several publicly available AMQP implementations and many of them are open-source products with support of C# language. More information about the protocol can be found at [AMQPWG09] and in [AMQPWG09a].

5.1.5 Concluding remarks

In previous sections we have introduced some of the most popular communication middleware models. Only the basic architecture of each model was described and related characteristics discussed. However, it should suffice to give some conclusion and choose the most suitable middleware for implementation of monitoring system. During the discussion it became apparent that each middleware has its positives and negatives and that it is important to select, during the design of any system, the most appropriate one for particular cases.

To summarize the previous text: At first Remote Procedure Call (RPC) was briefly described because many other middlewares, including the newest ones, use its model for communication and enriches it by some

other features. From group of object-oriented RPC-based middleware style, called Object Request Broker, three major implementations were discussed. .NET Remoting, which suffers from tight coupling, language and platform dependence, Common Object Request Broker Architecture, which has some advantages compared to .NET Remoting, such as high performance data distribution and many additional features presented in a form of optional services. However the main problem of CORBA is its complexity leading to long time learning curve and its missing built-in support of C# language. Even though there is an IIOP.NET project, which incorporates CORBA/IIOP support into .NET, leveraging the .NET Remoting framework, it does not suffice because the project is not very active today, same as development around the CORBA specification. The last protocol introduced within the ORB-style model was an Internet Communications Engine, which concept is very similar to CORBA, but is much simpler, easy to learn and use, very well documented and provides C# language API. In addition with many built-in services for different communication scenarios, Ice seems to be a very promising ORB-style middleware.

After few ORB middlewares, different style of communication was described. It was a Data Distribution Service based on a data-centric publish-subscribe model. Its big advantage is high-performance data distribution with ultra low latencies and ultra high throughput and support for many QoS policies, which allows configuring different aspects of data availability, delivery and timeliness. Unfortunately, as of today, there is no available open source implementation with support of C# language, however should be introduced in couple of months.

The last type of introduced middleware is a message-oriented one. Under this category two types of standards were described. The first one was Java Message Service, which is implemented by many vendors on the market and the second one was an open protocol called Advanced Message Queuing Protocol developed initially for finance sector. Both standards dispose of general advantages of message-oriented middleware, such as asynchronous communication, decoupled architecture and good options for data distribution through publish-subscribe mechanism. Both of them are implemented in open source projects, from which couple of them provides API for C# language. What differs JMS compliant implementations from AMQP implementations is that AMQP is open standard on the wire-protocol level, which brings the interoperability between different implementations from different vendors. The other fact, which should be noted here, is that AMQP is implemented with a focus on performance required in finance sector.

Message-oriented middleware was finally chosen as a most suited one for implementation of monitoring system, especially its AMQP-based implementation. The reason why it was preferred before Ice and DDS is that Ice is not free for commercial sector and DDS does not provide as of today C# language API. However both of them should be envisaged as possible alternatives for the case that MOM would not satisfy the requirements in the future. This should be kept in mind during the implementation and communication module should be utmost decoupled from other system implementation, so it could be relatively easy to change the communication middleware in the future.

5.2 Message-Oriented Middleware comparison

Message-oriented middleware was proposed in the previous sections as a most suitable one for monitoring system implementation and therefore in the following sections some of the most popular message-oriented middlewares will be introduced and their feature set described. At first we will focus on implementations based on JMS standard followed by AMQP compliant implementations and finally, based on the previously discussed features, the most suitable MOM implementation, from the monitoring system point of view, will be chosen.

5.2.1 Apache ActiveMQ

Apache ActiveMQ is the most popular and powerful open source message broker. It is developed and maintained by Apache Software Foundation [ASF09] and is released under the Apache 2.0 License. Its messaging engine is written in the Java programming language and fully implements version 1.1 of Sun Microsystem's Java Messaging Service standard. The project is developed for many years, the most recent stable version is ActiveMQ 5.2 released in November 2008 and it is continually supported and updated by the ASF and the user community. It provides many advanced features for various messaging pattern paradigms. Some of them are [ASF08]:

- **Message groups** - This nice feature ensures that all messages for the same message group will be sent to the same consumer while that consumer stays alive. As soon as the consumer dies, another one will be chosen. That brings a load balancing of the processing of messages across multiple consumers and high availability/auto-failover to other consumers if a JVM goes down.
- **Virtual topics** - With a virtual topic, a publisher publishes messages to the virtual topic just as it would with a regular JMS topic. However, the topic is presented to the consumer as a queue and not a topic. This gives the developer all advantages of queues such as a message grouping, load balancing, and failover. The queue also serves as a durable subscription because if the consumer becomes inactive, then after its reactivation it will receive any messages that were published to the topic while it was down.
- **Destination wildcards** - Provides easy support for federated name hierarchies, the popular concept in financial market data. It is used to organise events (such as price changes) into hierarchies, which then can be subscribed with usage of wildcards informing the broker in what information is the subscriber interested in.
- **Composite destinations** - Allows developer to map a single, virtual JMS destination to a collection of physical JMS destinations. In other words, the composite destination feature allows developer to send a message to multiple physical destinations (queues or topics) with a one single send operation.
- **Clustering** - ActiveMQ provides a set of clustering functions. It supports failover transport protocol to allow clients to automatically failover from one broker to another in case of broker failure, you can connect brokers to a network, which provides a store and forward mechanisms. For high availability of brokers one can use a master/slave configuration, which ensures that you get immediate failover to the slave in case of catastrophic hardware failure of the master's machine, file system or data centre.

Even though ActiveMQ is written in Java, with the primary goal of implementing the JMS API, ActiveMQ also supports other programming languages, e.g. C#, Ruby, Python, C/C++. It can be deployed on any operating platform (e.g., Windows, UNIX, and Linux) that provides a compatible Java Virtual Machine and supports many different low-level transport protocols such as TCP, SSL, HTTP, HTTPS, and XMPP. More information about the project can be found at [ASF08]

5.2.2 TIBCO Rendezvous

TIBCO Rendezvous is one of the leading commercial messaging products for real-time high throughput data distribution applications. It is developed by a software company TIBCO Software Inc [TIBCO09], which provides enterprise software that helps companies achieve service-oriented architecture. TIBCO Rendezvous is one of many products, which company offers. It is used by over 2000 companies worldwide

for applications ranging from market data distribution and trading applications to real-time control systems, transportation networks and military applications.

Major advantages of TIBCO Rendezvous are [TIBCO09a]:

- **Performance** - It is developed with a focus on high-performance ensuring ultra-low latencies at low microsecond levels.
- **Reliability** - Delivers proven reliability for 24x7 environments. It is deployed in production in many mission-critical systems as a messaging backbone.
- **High availability** - It has built-in support for fault tolerance and load balancing, including APIs that enable developers to build fault tolerance and load balancing into applications.
- **Large and active community** - Because the product is used by thousands of companies worldwide, the user community is large and very active.
- **Easy administration** - Provides low cost of administration through centralised management and self-managing protocols.

TIBCO Rendezvous supports all commonly used platforms and its APIs are available in Java, C, C++, C#, Perl, and COM. The last released version is TIBCO Rendezvous v6. More information about product can be found at [TIBCO09b].

5.2.3 FioranoMQ

FioranoMQ is one of the leading commercial messaging products on the market today developed by Fiorano Software Technologies P Ltd [Fiorano09]. This company was the first one to release a commercial JMS product in September 1998, when Sun Microsystems released the JMS standard, and has since maintained its lead over competitors. Among the most valuable features that product offers include:

- **Performance** - Designed for high-performance, low latency message delivery for demanding enterprise applications.
- **High availability** - Provides, besides the standard shared disk high availability options, a software-only high availability without imposing any specific hardware requirements. This provides applications with automatic fault-tolerance capabilities and allows them to focus on the business logic.
- **Clustering and load balancing** - It allows to run multiple server instances concurrently to provide increased scalability and reliability. Load balancing architecture involves the use of dispatcher component, which is connected to multiple FioranoMQ servers and is responsible to route incoming client connections to the least loaded server in a cluster [Fiorano09a].
- **Hierarchical topics** - Provides a logical correlation between topics, so the programmer can build hierarchical topic structures.
- **Durable connections** - Provides client applications with a fault-tolerance connection mechanism. If an application creates a durable connection, it need not worry about reconnecting back to the server in case of some fault. This is automatically handled by FioranoMQ's runtime library. If any message is sent during the disconnected phase, it is stored in a local repository in the client machine [Fiorano09a].

FioranoMQ is a proven standards-based messaging solution being used by over 300 companies worldwide. It supports all standard platforms and provides multiple native runtime libraries written in C,C++ and C# languages, which allow non-java applications to communicate directly with the java server. The most recent version is FioranoMQ 9.0.1 and more information about the product can be found at [Fiorano09a].

5.2.4 RabbitMQ

RabbitMQ is a complete open source implementation of AMQP, tracking the core features of the current specification. It is developed and supported by Rabbit Technologies [RabbitMQ09], a joint venture between UK-based LShift [LShift09], specialists in custom software for the telecommunications, retail and finance sectors, and Cohesive Flexible Technologies [CohesiveFT09], the provider of virtual appliances.

RabbitMQ server is written on top of the widely-used Open Telecom Platform [OTP09] originated and supported by Ericsson [Ericsson09]. The Open Telecom Platform (OTP) is used by multiple telecommunications companies to manage switching exchanges for voice calls, VoIP and now video. These systems are designed to never go down and to handle truly vast user loads. It is a battle-tested library of management, monitoring, and support code for constructing extremely high-performance, reliable, scalable, available (nine nines = 99.9999999%) distributed network applications [RabbitMQ09]. OTP is written in Erlang [Ericsson09a], which is general-purpose programming language and runtime environment with built-in support for concurrency, distribution, fault tolerance and incremental code loading [Ericsson09a].

Major advantages of RabbitMQ implementation of AMQP protocol are:

- **Reliability** - As already mentioned, RabbitMQ is written on a top of production tested and reliable Open Telecom Platform.
- **Strong community** - There are many messages per month on mailing list and key developers respond frequently. No single thread is ignored.
- **Active development** - RabbitMQ is under very active development. New versions with new features and bug fixes are releasing very frequently.
- **Sufficient performance** - Open Telecom Platform is constructed for high-performance applications. Thousands of messages per second should not be problem.

As the server component of RabbitMQ is written in Erlang, it is platform-neutral and runs on wide range of operating systems. It provides many language-specific clients, such as Java, C#, Ruby, Python and few others and implements AMQP protocol version 0.8 and 0.9. It is distributed under the open-source Mozilla Public License and the most recent version is RabbitMQ 1.6.0 released in June 2009. More information about the project can be found at [RabbitMQ09].

5.2.5 Apache Qpid

Apache Qpid is an open source messaging solution which implements the Advanced Message Queueing Protocol. It is developed and maintained by Apache Software Foundation [ASF09] and released under the Apache License Version 2.0. Apache Qpid provides two AMQP message brokers, one written in Java and one written in C++. The C++ broker appears to be where the good stuff is at. Among the most valuable features that product offers include:

- **Performance** - It is optimized for high performance and low latencies. C++ broker and client supports RDMA transport to bring an additional performance boost and cut off latencies to 50us per round trip.
- **Latest AMPQ specification** - Apache Qpid is the most aggressive project in implementing the latest version of the AMQP specification and provides its most complete and compatible implementation.
- **Custom exchanges** - C++ broker supports additional XML Exchange for XPATH query routing and custom exchanges can be developed via plug-ins to provide own custom routing logic and algorithms.
- **Clustering and fault tolerance** - The C++ broker has plug-ins for Active-Active clustering which keep all the nodes of the cluster in sync. This means that any action that is performed on one of the brokers on the cluster is performed on all of them at the same time. New nodes can be added to the cluster at any time, and removed at any time with no consequences. Client can dynamically track the nodes in the cluster and reconnect as required.
- **Broker federation** - Federation provides the ability to create networks of brokers that communicate with each other in all types of topologies. This allows a producer to publish messages to one broker and someone to consume the messages from another broker somewhere on the broker federated network.
- **Custom messaging patterns** - The C++ broker supports ring queues, last value queues and initial value caches on exchanges.

Apache Qpid provides AMQP Client APIs for C++, C#, Ruby, Python and Java. The C++ broker supports AMQP 0-10 and runs on Linux systems and on Windows, while Java broker runs on any Java platform and supports AMQP/0-8 and AMQP/0-9. The latest release Apache Qpid is version 0.5 from January 2009. More information about project can be found at [ASF09b].

5.2.6 OpenAMQ

OpenAMQ is an open source and free to use message broker written in C and released under the terms of the GNU General Public License. It is developed by iMatix Corporation [iMatix09], which stands at the beginning of AMQP protocol specification and the main features of broker are:

- **Performance** - It is designed to be extremely robust and fast. To achieve the best performance, OpenAMQ does not implement everything in AMQP standard and drop some features, which could decrease the performance of whole solution. At the same time it supports Direct Mode, which runs over Direct Messaging Protocol for specific kinds of high-speed applications. This method allows transferring messages much faster. The speed improvement comes from a simpler wire-level encoding for messages, message batching, and the ability to push messages through the OpenAMQ server with less processing. Client applications also benefits from reduced CPU usage.
- **Failover support** - OpenAMQ's failover model consists of two dedicated servers (OpenAMQ server processes) in a primary-backup pair. At any given time, one of these accepts connections from client applications (it is the "master") and one does not (it is the "slave"). Each server monitors the other. If the master disappears from the network, the slave takes over as master.
- **Federation support** - OpenAMQ's federation model lets architects build networks of OpenAMQ servers that implement specific kinds of message flows. This brings the ability to build very high-performance

pub-sub architectures for scenarios with high volumes of data going to very many subscribers and to partition a large network geographically. Unfortunately, there is no support to use federation and failover concurrently.

OpenAMQ implements AMQP/0.9 and AMQP/0.9.1, is built on a high-performance portability framework written in ANSI C, which is portable to Linux, Windows, Solaris, AIX, OS/X and other Unix systems. It provides language bindings to C, C++, Java JMS, Python and Ruby. C# language binding is not provided, but AMQP-compliant C# clients from other vendors could be used to interact with OpenAMQ broker. The most recent stable and recommended version is OpenAMQ/1.3 released in May 2009. More information about project can be found at [iMatix09a].

5.2.7 Conclusion

Previous sections briefly described the most popular implementations of message oriented middleware and their most important features. Even though it was decided that the monitoring system should be based on an open source software only, two commercial MOM implementations were mentioned too, to let the reader know, which commercial middleware can be used to replace the open source candidate in the future in case of any problems. The following paragraph will discuss open source candidates and choose the most suitable one for monitoring system implementation.

During the evaluation of all four open source candidates it turned out that not all of them are stable enough. The OpenAMQ broker failed very frequently in case of persistance queues usage and therefore was at early stage of testing completely removed from the list of possible candidates. The remaining three implementations were a JMS-based ActiveMQ and two AMQP standard compliant solutions, the Apache Qpid and RabbitMQ. The evaluation was at first focused on AMQP implementations especially because of the AMQP's interoperability and because the standard was originally designed for an usage in finance sector. Both implementations proved their very good stability during the general testing and sufficient performance for needs of monitoring system. The main difference between them was especially in a provided documentation and community support. Apache Qpid's documentation is not very detailed and therefore the learning curve was much longer than in case of RabbitMQ. RabbitMQ provides a very nice documentation and its community is one of the best I've ever seen. This fact was the main reason why the RabbitMQ was finally preferred before the Apache Qpid in case of AMQP protocol implementations.

The last evaluated middleware implementation was already mentioned ActiveMQ and was directly compared with RabbitMQ implementation. It have to be noted here that ActiveMQ is a very popular and by many years proven JMS implementation. Its community is also very strong and the documentation doesn't lack any important part. Despite this fact the RabbitMQ's documentation and community is still a little bit further. The other important thing, which appears during the evaluation of ActiveMQ is that the C# language binding called NMS is not stable enough in case of larger data volumes and its performance is also much worse than what states in ActiveMQ's specification. It turns out, that the problem is really in a C# language binding only and that the native java client do not suffer from these issues. Therefore another experiment (based on a recommendation from ActiveMQ's developer forum) was done: the java client library was compiled with an IKVM.NET project [IKVM09], which is an implementation of Java for Mono and the Microsoft .NET Framework and is useful when the developer needs to use java libraries in .NET applications. This compiled library provided contrary to NMS stable results and the data distribution was also much faster. However because the solution with compiled java library into a .NET one is not supported by developers of ActiveMQ at all and because the RabbitMQ's evaluation was completely without any

problems, have the strongest community and is under a very active development, the decision was made to choose RabbitMQ implementation as a most suitable middleware for monitoring system.

Chapter 6

Modularity

One of the very important requirements is a modular design of whole monitoring system, so it can be easily extended in the future. Therefore it is necessary to split the system into independent modules as much as possible. To be able to do so, the implementation of the system should be based on some plugin framework, which is easy to use and provides sufficient functionality for future extensions. There are two ways, the programmer can take, the first way is to implement such framework himself or to use an existing solution. Because implementation of plug-in framework, which has sufficient functionality, is not an easy task and would take a lot of effort, the second way was chosen. If we take a look at existing, open source plug-in frameworks for .NET language, we find that there are only few of them, which are worth mention.

- **System.Addins** - Framework, which is part of .NET Framework 3.5. It provides a robust plugin loading framework with options of loading assemblies with different levels of trusts. It is primarily code-based and complex solution. For more information about namespace please see [Microsoft09b].
- **Managed Extensibility Framework** - Framework developed by Microsoft Corporation engineers, which should be simpler and more accessible alternative to System.Addins and should be a part of .NET Framework 4.0 when released. Current stable version is MEF Preview 6 from 13.7.2009. For more details about framework see [Mef09].
- **SharpDevelop Core** - Plug-in management framework, which is a part of SharpDevelop IDE implementation and provides the ability to declare extension endpoints and then discover and load plugins that service those endpoints. The plug-in infrastructure is licensed under LGPL, and so can be used in commercial closed-source solutions as well. More information about SharpDevelop and its plug-in infrastructure can be found at [SharpDev09].
- **Mono.Addins** - Project derived from MonoDevelop IDE [MonoDev09], which was designed for C# and other .NET languages and enables developers to write applications under Linux, Mac OSX and Windows. Mono.Addins is a framework for creating extensible applications basically in Mono environment, but can be used for standard .NET applications as well. For more information see [MonoAddins09].

From deeper look into above mentioned frameworks and its functionalities, the following conclusion was made. The positive thing about System.Addins is that it is a part of .NET framework itself, so there is no need to use third party libraries and you can rely on future support by Microsoft. It seems to be very solid framework for extensibility, but unfortunately it is quite complicated as well and is more suitable for

large enterprise wide applications. The Managed Extensibility Framework is simpler and more accessible alternative to System.Addins and because it is developed by Microsoft too and should be part of .NET framework in the future, it seems to be a better candidate than System.Addin namespace. Unfortunately the project in its current phase is still pretty fuzzy. There is not much documentation yet and few examples, which are available do not explain the details very well. SharpDevelop core library is on the other hand quite well documented, provides great functionality in a form of extension points and a couple of good examples how to use its features.

The last, but not least Mono.Addins project is quite similar to SharpDevelop in a way of extension points declaration, conditional plug-in loading etc., but it appears, that it was influenced by System.Addins as well. It seems to provide the basics from System.Addins namespace, but with less sophisticated options for plugin management. It brings more simplicity with attribute-based registration and also supports the XML manifests and the infrastructure for online plug-in repositories. Because Mono.Addins project besides its simplicity and wide range of functionality has a pretty good FAQ, documentation and reference manual and because it provides a robust set of examples that help user to paint a picture of how to develop applications with extensible architecture, it was chosen as a most suitable candidate for monitoring system implementation. To better understand how monitoring system plug-in architecture works, please see Mono.Addins project's documentation page at [MonoAddins09a].

Chapter 7

Application Framework

Because a lot of required functionality of monitoring system is general and not specific for monitoring only, it was decided to implement an application framework first. The framework encapsulates all this general functionality and can be further used and extended by other applications, not only for monitoring purposes. If you think about this, you realize that such solution moves the abstraction level one step further. It allows to easily integrate a set of applications (modules) together with a basic knowledge of application framework. Let's say we have three applications, which can be used independently, but it would be nice if they can communicate together as well. Such applications can be monitoring system used for monitoring electronic trading infrastructure, trading platform used by traders for working client orders and monitoring system for market data flow. Each of mentioned system can be used as a separate application, but much more interesting solution is to allow to use them together. As an example scenario it can be mentioned a usage of electronic trading monitoring system and market data monitoring system together to provide the end user ability to see what was the exact situation on current instrument, when the client's order was received. The first part of this chapter describes the implementation of such application framework and explains, how it can be used for application integration and what are the main advantages it brings. The second part will then focus on implementation of monitoring system, which was developed as an extension module to the application framework, itself.

As discussed in Chapter 4, the multi-tier client-server architecture is the most suitable one for monitoring system implementation. The same applies for application framework and its architecture overview is illustrated in Figure 5.8.

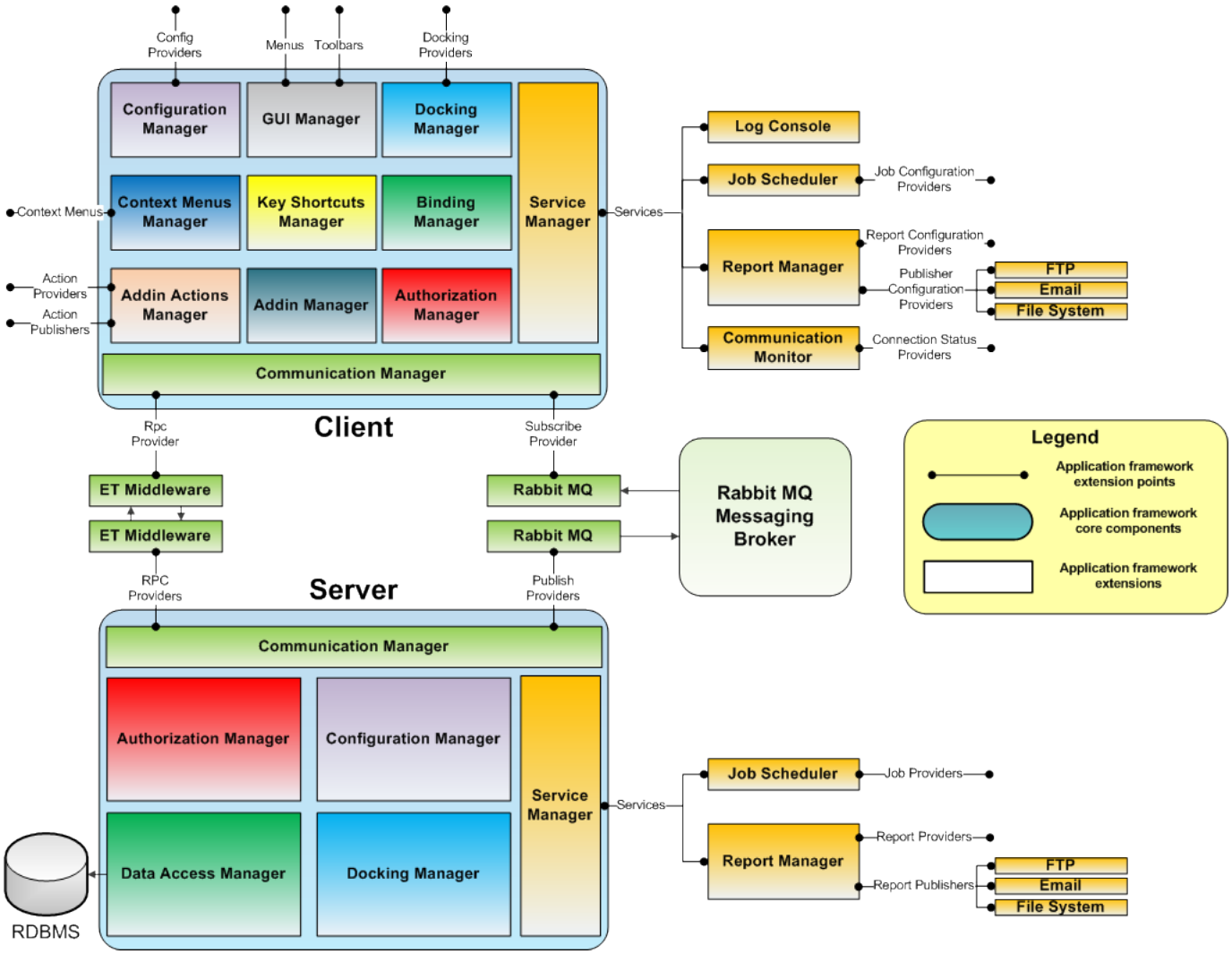


Figure 7.1: Application framework architecture

As you can see, the application framework consists of server and client part and both include a set of components and expose a set of extension points. Components can be divided into three categories, server side specific component, client side specific component and components, which are used on both client and server side and are split into server, shared and client libraries. The same rule applies to extensions too. All libraries and classes are logically split into three different namespaces, the `WAC.ET.Framework.Server` namespace for server side implementation, `WAC.ET.Framework.Shared` namespace including common libraries for both server and client side and `WAC.ET.Framework.Client` namespace, which consists of libraries used for client side implementation. Before we describe client and server side implementation in more detail, we will focus on communication layer, which is present on both client and server side and is fundamental for interaction between clients and server.

7.1 Client-server communication

In Chapter 5 we have deeply discussed the selection of communication middleware, because it is a very important part of each distributed system. Because each middleware has its pros and cons, the communication layer of application framework was implemented as an extensible library, so it is not tightly coupled to a one specific middleware. Such implementation brings the following advantages:

- **Very easy middleware replacement** - In case that a new middleware emerge on the market, which brings new possibilities like better performance, better fail-over, better scalability etc., it should be really easy to replace the currently used middleware with a new one. The only thing, which has to be done is to implement an extension, which uses for communication this new middleware and plug this extension to the system.
- **Ability to use different middlewares for different types of communication** - It is common scenario, that different middlewares are suitable for different types of communication. Therefore the application framework exposes extensions for two communication channels, one is RPC channel used for so called request-response communication and the second one is pub-sub channel used for publish-subscribe communication. Each channel then can use the most suitable middleware for given communication method.
- **Ability to use more than one middleware for one type of communication concurrently** - This is useful in such scenarios where there is a need to run client applications on different platforms and currently used communication middleware do not support all of them. Because the server-side communication layer expose extension points on which can be bound more than one extension, it is quite simple to plug in another extension, which supports the required platform and can communicate with clients, which use the same client side extension.

The Figure 7.2 illustrates the communication example, which benefits from all three above mentioned advantages.

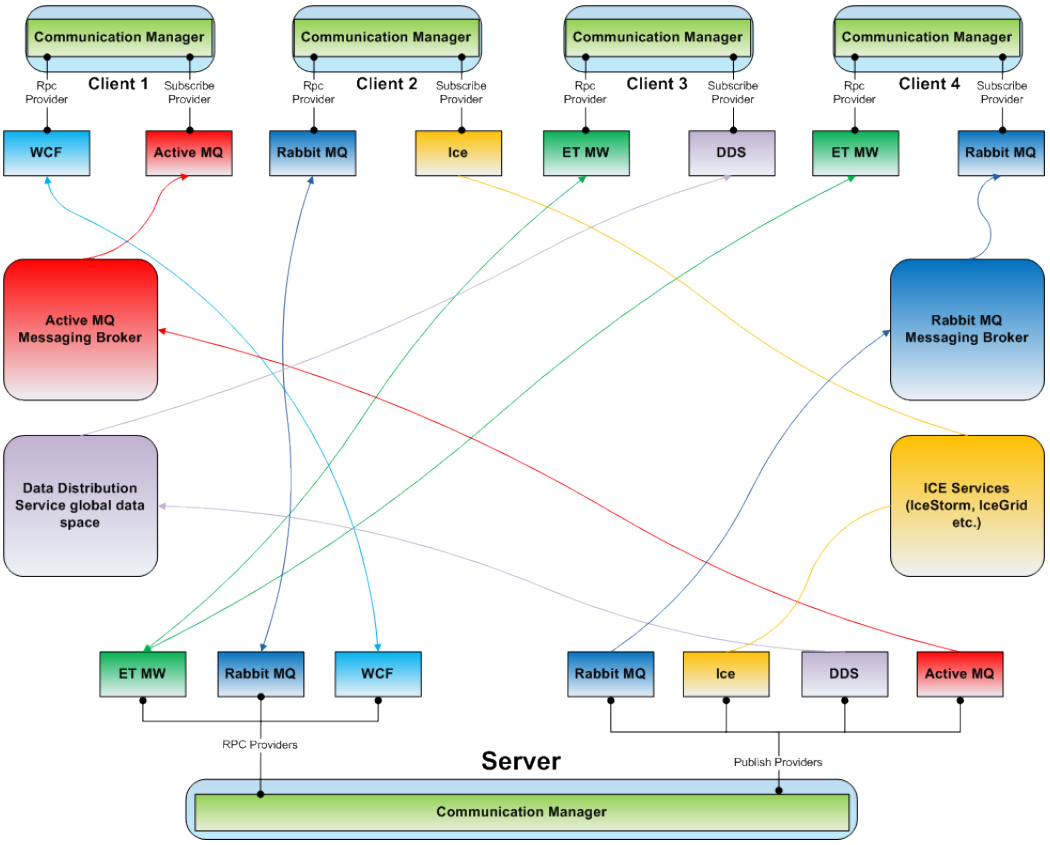


Figure 7.2: Application framework communication example

As you can see, there are four clients connected to the server and each client uses a different communication extension pair. Client 1 uses WCF for RPC communication and ActiveMQ for subscriptions, whilst Client 2 uses RabbitMQ for RPC communication and Ice for subscriptions. Clients 3 and 4 are using different extensions too. Whilst on client side, only one extension can be bound to each communication channel, on the server side, the number of extensions per channel is not limited. In the above illustrated scenario there are three communication extensions for RPC channel (Electronic Trading Middleware, RabbitMQ and WCF) and four extensions for pub-sub channel (RabbitMQ, Ice, Data Distribution Service and ActiveMQ). Each extension is than responsible for communication with clients, which use the same extension on same communication channel. For better understanding how can be the communication layer such modular, the following paragraph briefly explains the communication process.

During the start-up process of application framework server, available communication extensions are discovered, loaded into application domain, initialized and all RPC providers starts to listen for incoming requests and all publish providers open channel for publishing server events to currently connected and interested clients. Each extension implements an interface specific for the type of communication channel, so general public methods can be called from application core without knowing a specific implementation of underlying provider. All loaded extensions are then stored in the wrapper class, which is the only entry point used by the other server components for communication. Because each client, which want to communicate with the server can have a different pair of communication provider extensions, it is important to send in a logon request a description of client side communication configuration, so the server knows, what communication extension have to be used for interaction with currently connected client in the future. For describing the client side configuration two abstract classes from `WAC.ET.Framework.Shared.-`

`Core.Communication` namespace are used. The first is `ClientRpcInfo` for describing a currently used RPC communication channel and the second one is `ClientPublishInfo` used for describing a currently used pub-sub communication channel. Because both classes are abstract, each client side extension have to provide classes, which derives from these abstract ones and which could carry additional information specific for currently used communication channels.

For RPC communication, some additional rules applies too. Each RPC request has to derive from the `RpcRequest` and each RPC response has to derive from `RpcResponse` class. Both classes are abstract and also present in `WAC.ET.Framework.Shared.Core.Communication` namespace. These two RPC communication base classes are important for easier request handling on server side. `RpcRequest` class wraps the request unique identification, so there is no need to implement this on each derived request. This unique identification is used on server side for finding the method, which should be called to process the request and return the request's result. Each component on server side can then register a method delegate to such request identifier. The publish process from server to all interested clients is simple too. Because server knows communication configuration of all connected clients (it was published to server with the client's logon request), then whenever there occurs a server event, which should be published, it is passed to a mentioned server communication wrapper class and this class finds all publish provider extensions, to which the given event should be passed and published by them to clients. More details, such as which concrete classes are used in server and client side communication process and how to implement the communication providers, will be described in communication sections of both client and server implementations.

In next few sections, core components on both server and client side will be described. These sections explain what is the main purpose of each component, how they work and in case they expose some extension points, how they can be extended in the future.

7.2 Server implementation

Application framework server (in the following paragraphs simply called as a server) wraps the data access layer and business layer and its main purpose is to centralise data storage, improve the maintenance and scalability of whole architecture and to provide functionality common for all clients, so the client implementation can be as lightweight as possible. The architecture of the server is outlined in Figure 7.3.

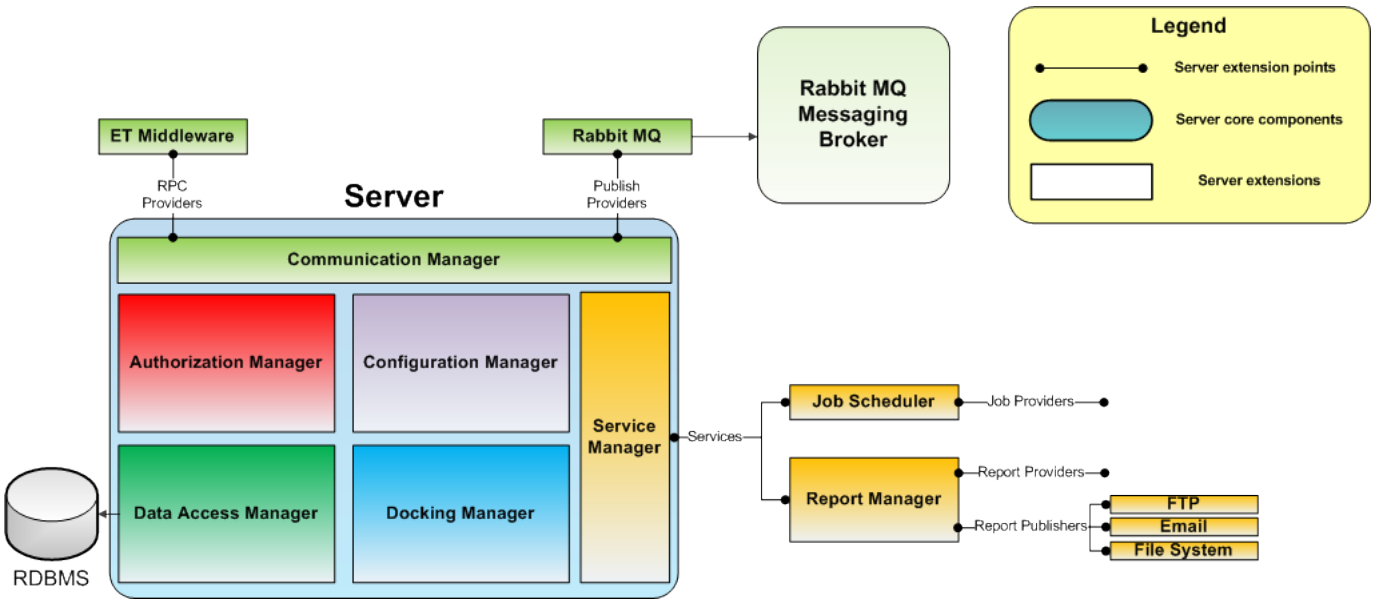


Figure 7.3: Application framework server architecture

As you can see, there are a couple of core components, some of which expose extension points, so they can be extended by external modules in the future and few of them are already present in the current implementation. All server side libraries and classes are part of `WAC.ET.Framework.Server` namespace. The core namespace, which encapsulates all core server components is `WAC.ET.Framework.Server.Core`. Beside many other classes, it includes the `ServiceManager` responsible for loading external server modules, `ConfigurationManager` responsible for storing, loading and providing configurations to clients, `AuthorizationManager`, that is used for client authorization and authentication and for permissions management, `DockingManager` responsible for storing, loading and providing application layouts to clients and `CommunicationManager`, which is responsible for loading underlying communication providers and which wraps the whole communication process with clients into the single entry point. The `WAC.ET.Framework.Server.Core` library is defined in Mono.Addins framework as a main add-in root, so it can be extended by other modules.

As already mentioned, the component responsible for loading external server modules is `ServiceManager`. It is the first component, which is initialized after the server start-up process and during its initialization it discovers and loads all external modules, which extend the *Services* extension point. The declaration of this extension point is shown in Example 7.1.

Example 7.1 Services extension point declaration

```
<ExtensionPoint path = "/Server/Services">
  <ExtensionNode name="Service" type="WAC.ET.Framework.Shared.Core.Common.ServiceExtensionNode">
    <Description>Registers a new service, which can be used by other plugins.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

The `ServiceExtensionNode` can be described with following attributes:

- `order` - Optional attribute through which you can specify the ordering in which all loaded modules will be initialized and started.

- `configFileName` - Optional attribute through which you can specify a service configuration file name path. If specified, the extension node will automatically read the content of the configuration file and pass it to the underlying extension implementation.
- `type` - Required attribute telling the extension node, what is the type of service extension, so the extension node can dynamically activate a new instance of service of given type. Each service extension instance has to implement an `IComponent` interface from `WAC.ET.Framework.Shared.Core.Common` namespace and its declaration is shown in Example 7.2.
- `isRequired` - Required attribute defining if the service extension is required for correct application run or not. In case a failure occurs during required service initialization or start-up, the `ServiceManager` stops with processing of further services, logs the failure and terminates the application.

Example 7.2 IComponent interface declaration

```
public interface IComponent
{
  /// <summary>Event which occurs when state of the component changed.</summary>
  event ComponentStateChangedDelegate ComponentStateChangedEvent;

  /// <summary>Gets component identifier.</summary>
  string ComponentIdentifier { get; }

  /// <summary>Gets indication whether the component is initialized or not.</summary>
  bool IsInitialized { get; }

  /// <summary>Gets state of the component.</summary>
  ComponentState State { get; }

  /// <summary>
  ///  Gets indication whether the component's state is OK or whether there is a problem.
  /// </summary>
  /// <remarks>Component's state indication.</remarks>
  bool StateOk { get; }

  /// <summary>Gets human readable string describing component state.</summary>
  string HumanReadableStateInfo { get; }

  /// <summary>Initializes the component.</summary>
  /// <param name="configuration">Configuration string.</param>
  void Initialize(string configuration);

  /// <summary>Starts the component.</summary>
  void Start();

  /// <summary>Stops the component.</summary>
  void Stop();

  /// <summary>Unitializes the component.</summary>
  void Uninitialize();
}
```

7.2.1 Communication

This section describes the most important parts of server communication layer implementation and how it can be extended by new communication providers in the future. There are two extension nodes defined, `RpcClientCommunicationProviderExtensionNode`, which is used for registration of RPC communication provider extensions and `PublishClientCommunicationProviderExtensionNod-`

e, which is used for registration of publish communication provider extensions. XML manifest declaration for both extension points is shown in Example 7.3.

Example 7.3 Rpc and publish communication provider extension points declaration

```
<ExtensionPoint path = "/Server/Communication/Client/RpcProviders">
  <ExtensionNode name="RpcClientCommunicationProvider"
  type="WAC.ET.Framework.Server.Core.Communication.RpcClientCommunicationProviderExtensionNode">
  <Description>
    Registers a new communication provider for rpc communication with client systems.
  </Description>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Server/Communication/Client/PublishProviders">
  <ExtensionNode name="PublishClientCommunicationProvider"
  type="WAC.ET.Framework.Server.Core.Communication.PublishClientCommunicationProviderExtensionNode">
  <Description>
    Registers a new communication provider for publish-subscribe communication with client systems.
  </Description>
  </ExtensionNode>
</ExtensionPoint>
```

Both extension nodes have similar attributes, that describes the extension and are as follows:

- `configFileName` - Optional attribute through which you can specify a communication extension configuration file name path. If specified, the extension node will automatically read the content of the configuration file and pass it to the underlying extension implementation.
- `name` - Required attribute providing the name of communication provider.
- `type` - Required attribute telling the extension node, what is the type of communication provider extension, so the extension node can dynamically activate a new instance of underlying provider. Each RPC provider extension instance have to implement an `IRpcClientCommunicationProvider` interface shown in Example 7.4 and each publish provider extension have to implement `IPublishClientCommunicationProvider` interface described in Example 7.5.

Example 7.4 IRpcClientCommunicationProvider interface declaration

```
public interface IRpcClientCommunicationProvider
{
  /// <summary>Occurs whenever the one of currently connected clients disconnects.</summary>
  event ClientDisconnected OnClientDisconnected;

  /// <summary>Gets the name of provider.</summary>
  /// <value>The name of provider.</value>
  string Name { get; }

  /// <summary>Called when the client connection was approved.</summary>
  /// <param name="rpcInfo">Description carrying the RPC communication
  ///                       parameters, which should be used for approved connection.</param>
  void ClientConnectionApproved(ClientRpcInfo rpcInfo);

  /// <summary>Called when the client connection was removed.</summary>
  /// <param name="rpcInfo">Description carrying the RPC communication
  ///                       parameters, which were used for removed connection.</param>
  void ClientConnectionRemoved(ClientRpcInfo rpcInfo);

  /// <summary>
  ///  Adds a request delegate, which is responsible for handling the requests with given identifier.
  /// </summary>
  /// <param name="requestId">      Request identifier.</param>
  /// <param name="requestDelegate">Delegate responsible for handling the requests with given
  ///                               identifier.</param>
  void AddRequestDelegate(string requestId, RequestDelegate requestDelegate);

  /// <summary>Initializes the communication provider.</summary>
  /// <param name="configuration">The configuration string.</param>
  void Initialize(string configuration);

  /// <summary>Starts the communication provider.</summary>
  void Start();

  /// <summary>Stops the communication provider.</summary>
  void Stop();

  /// <summary>Uninitialize the communication provider.</summary>
  void Uninitialize();
}
```

Example 7.5 IPublishClientCommunicationProvider interface declaration

```
public interface IPublishClientCommunicationProvider
{
  /// <summary>Gets the name of provider.</summary>
  /// <value>The name of provider.</value>
  string Name { get; }

  /// <summary>Called when the client connection was approved.</summary>
  /// <param name="publishInfo">Description carrying publish communication
  ///                           parameters, which should be used for approved connection.</param>
  void ClientConnectionApproved(ClientPublishInfo publishInfo);

  /// <summary>Called when the client connection was removed.</summary>
  /// <param name="publishInfo">Description carrying publish communication
  ///                           parameters, which were used for removed connection.</param>
  void ClientConnectionRemoved(ClientPublishInfo publishInfo);

  /// <summary>Publishes the given message of given publish type to all interested clients.</summary>
  /// <param name="group">Enumeration indicating the type of published message.</param>
  /// <param name="message">The message to be published.</param>
  void Publish(PublishGroup group, object message);

  /// <summary>Initializes the communication provider.</summary>
  /// <param name="configuration">The configuration string.</param>
  void Initialize(string configuration);

  /// <summary>Starts the communication provider.</summary>
  void Start();

  /// <summary>Stops the communication provider.</summary>
  void Stop();

  /// <summary>Uninitialize the communication provider.</summary>
  void Uninitialize();
}
```

Class responsible for loading extension nodes from above mentioned extension points is `ClientCommunicationManager`. Beside the loading, initializing and starting of the communication providers, it is responsible for passing request delegates and published messages to underlying communication providers, so other server components do not have to know, what communication providers are currently loaded and it is the only class, which these components have to know. If any server component needs to register its method as a specific RPC request handler, it calls the manager's `AddRequestDelegate` method and manager passes the given delegate to all underlying RPC providers. The same applies for event publishing mechanism, so if any server component have to publish the event to connected clients, it calls the manager's `Publish` method and communication manager passes the given event to all underlying publish providers. This brings the simplicity to communication layer usage within the server. As shown in Figure 7.3, the current implementation of application framework server includes two communication extensions. As a RPC communication provider extension is used electronic trading middleware, which was developed internally in sponsoring company and a publish communication provider extension is implemented using RabbitMQ middleware described in Section 5.2.4.

7.2.2 Job scheduler

Job scheduler is an external application framework module bound to the service extension point and provides the functionality to schedule the execution of different types of jobs on regular basis. It's implementation uses a Quartz.NET framework[Quartz10], which is a feature-rich, open source job scheduling system

licensed under the terms of the Apache License, Version 2.0. It is a port of very popular java scheduling framework called Quartz and is written entirely in C# language. It has quite nice API documentation and tutorial, is very stable and provides a comprehensive list of features. Even though the description of Quartz.NET framework is out of the scope of this thesis, few basic classes, which are used by application framework later, will be briefly described in the following paragraph, so the reader can get a better picture of how the job scheduler component works.

Besides the many other useful classes and interfaces in Quartz.NET framework, the most important ones are `IScheduler` and `IJob` interfaces and `JobDetail` and `Trigger` classes. `IScheduler` is a core component, which is responsible for `JobDetail` and `Trigger` management. During it's initialization it can use a list of properties to describe the scheduler behavior, such as number of threads used to job triggering, type of job storage, type of scheduler instance and many others. To make a component executable by the scheduler, the component has to implement an `IJob` interface, which defines the only one `Execute` method. This method is called by scheduling framework whenever the component is triggered and the `JobExecutionContext` object that is passed to this method carries the run-time information about the job instance. This information contains a handle to the scheduler instance that executed the given job, the job's `JobDetail` object, a handle to the trigger, which fires the job and a few other items. At the time the job is added to the scheduler, a new `JobDetail` object is created by the Quartz.NET client. It is used for setting the various properties of the job, as well as a `JobDataMap` instance, which can be used to store a state of a given instance of the job class. For execution of jobs, the `Trigger` objects are used. Before the trigger can schedule a job, it has to be instantiated and its scheduling properties have to be set. Each trigger may also have a `JobDataMap` associated with them. This provides the ability to override job's parameters with values, which are specific to the firings of the trigger. Quartz.NET framework includes a set of different trigger types from which the most commonly used are `SimpleTrigger` and `CronTrigger`. These are the basic and most important classes in Quartz.NET framework and should be sufficient for understanding the following paragraphs about implementation and extensibility of application framework's job scheduler component. More information about Quartz.NET framework can be found at [Quartz10].

The core class of application framework's job scheduler component is a `SchedulerManager`, which is a wrapper of `IScheduler` instance and is responsible for managing the scheduler jobs. During the initialization, the `SchedulerManager` loads and initializes all available `SchedulerJobProviderExtensionNode` nodes for current application configuration. After that it loops through all loaded job providers, checks if there are some new jobs, which are not stored in database yet and if so, it adds these jobs to a list of supported jobs. At the same time it also checks for jobs, which are currently stored in database, but it's corresponding extension node, which should provide these jobs is not loaded. Such jobs are then paused for current application run, because without related job provider they can't be executed.

As already mentioned, the job scheduler component can be extended by new `ISchedulerJobProvider` extensions. For this is used a `SchedulerJobProviderExtensionNode` class defined in scheduler's XML manifest extension point as shown in Example 7.6.

Example 7.6 Scheduler's job providers extension point declaration

```
<ExtensionPoint path = "/Scheduler/JobProviders">
  <ExtensionNode name="JobProvider"
    type="WAC.ET.Framework.Server.Scheduler.SchedulerJobProviderExtensionNode">
    <Description>Registers a new job provider for task scheduler.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

The only attribute of this extension node is a type of registered `ISchedulerJobProvider` instance,

so it can be dynamically instantiated and initialized afterwards. The purpose of job provider is to provide a list of supported jobs, which can be scheduled and executed by scheduling framework. Each job should be provided as a pair of `JobDetail` instance and it's corresponding job type, which implements a `IJob` interface. At the same time, the `JobDetail` instance type has to be either a `StatefulJobWrapper` or `JobWrapper`. These two wrapper classes are defined in `WAC.ET.Framework.Shared.Scheduler` namespace and are responsible for carrying a specific `IJob` instance and its given execution state. When the scheduler framework executes a wrapper class, the specific `IJob` instance is unwrapped by `JobExecutor` helper class and its `Execute` method is called with job's execution context. Current the application framework server implementation provides one job provider extension, which is a part of external reporting module discussed in the following section.

7.2.3 Reporting

Same as job scheduler discussed in previous section, the reporting module is implemented as an external application framework module bound to the service extension point. Its main purpose is to provide a common functionality for generation and distribution of various application reports. The core class of module is `ReportManager`, which is the type bound to service extension point and therefore instantiated by `ServiceManager` instance during server startup. There are two types of extension points exposed by reporting module, for their declaration and corresponding extension paths please see Example 7.7. The first is for loading and initializing extensions of type `ReportProviderExtensionNode` and the second one is used for loading and initializing `ReportPublisherExtensionNode` types.

Example 7.7 Reporting module extension points declaration

```
<ExtensionPoint path = "/Reporting/ReportProviders">
  <ExtensionNode name="ReportProvider"
    type="WAC.ET.Framework.Server.Reporting.ReportProviderExtensionNode">
    <Description>Registers a new report provider to a report manager.</Description>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Reporting/ReportPublishers">
  <ExtensionNode name="ReportPublisher"
    type="WAC.ET.Framework.Server.Reporting.ReportPublisherExtensionNode">
    <Description>
    Registers a new report publisher, which can be used for publishing the reports.
    </Description>
  </ExtensionNode>
</ExtensionPoint>
```

Both extension nodes have same attributes, a `configFileName` attribute used to specify a configuration file, which should be used during extension node initialization and a `type` attribute telling the node, what type is the underlying provider, that should be instantiated and initialized. These two underlying provider types have to implement either `IReportProvider` or `IReportPublisher` interface and their declarations are shown in Example 7.8 and Example 7.9.

Example 7.8 IReportProvider interface declaration

```
public interface IReportProvider
{
  /// <summary>Gets the name of the provider.</summary>
  /// <value>Provider's name.</value>
  string Name { get; }

  /// <summary>Gets a list of the supported report types.</summary>
  /// <value>A list of the supported report types.</value>
  List<string> SupportedReportTypes { get; }

  /// <summary>Creates a report based on given configuration.</summary>
  /// <param name="configuration">The report configuration.</param>
  /// <returns>Information about report.</returns>
  ReportInfo CreateReport(string configuration);

  /// <summary>Initializes the provider.</summary>
  /// <param name="configuration">The configuration.</param>
  void Initialize(string configuration);

  /// <summary>Uninitialize the provider.</summary>
  void Uninitialize();
}
```

Example 7.9 IReportPublisher interface declaration

```
public interface IReportPublisher
{
  /// <summary>Gets the name of the publisher.</summary>
  /// <value>Publisher's name.</value>
  string Name { get; }

  /// <summary>Gets a list of the supported publish types.</summary>
  /// <value>A list of the supported publish types.</value>
  List<string> SupportedPublishTypes { get; }

  /// <summary>Initializes the publisher.</summary>
  /// <param name="configuration">The configuration.</param>
  void Initialize(string configuration);

  /// <summary>Publishes the given report with a way based on the given configuration.</summary>
  /// <param name="report">       The information about report.</param>
  /// <param name="configuration">The publish type configuration.</param>
  void Publish(ReportInfo report, string configuration);

  /// <summary>Uninitialize the publisher.</summary>
  void Uninitialize();
}
```

The `IReportProvider` extensions are used to tell the server, what reports are currently supported by application and are responsible for their creation. Each report has to be described by configuration class, that derives from `ReportConfiguration` abstract class and is passed in a serialized form to `IReportProvider`'s `CreateReport` method. Report provider creates a report based on passed configuration and returns the `ReportInfo` instance, which carries a report's name and its content and is passed to a corresponding `IReportPublisher` extension that is responsible for distribution of reports.

Besides the `ReportManager` class, which is responsible for loading and initializing above mentioned providers, the `WAC.ET.Framework.Server.Reporting` namespace includes a `JobProvider` class, that implements in previous section mentioned ISchedulerJobProvider interface and is used as an extension

of job scheduler component. The `JobProvider` instance provides only one job of type `GenerateReportJob`, that calls during its execution a `ReportManager`'s `GenerateReport` method with a corresponding `ReportJobConfiguration` instance carrying a description, what report type have to be created and by which report publisher the report should be distributed. This process allows the server to schedule a report creation and distribution by scheduling framework. The application framework currently includes three report publishers, which provides the ability to distribute reports to a FTP server, to a file system or through an email service.

7.3 Client implementation

Client of application framework (in next few sections simply called as a client) was developed especially to provide a common user interface and application control, so the end user does not have to learn new processes with each newly added module. This is very important, because the learning curve to use new module functionality is significantly reduced to a module's specific features and end user can start using the module almost immediately. The other very important advantage, which client provides, is an easy integration of external modules that extend the application functionality, so the module developers do not have to re-implement all general functions and can concentrate directly on features specific for a newly added module. This also significantly reduces the time needed to implement such module. The last, but not least advantage of client is the ability to allow cooperation between individual independent modules, which brings the possibility to create complex and flexible application environments.

The architecture of the client is outlined in Figure 7.4.

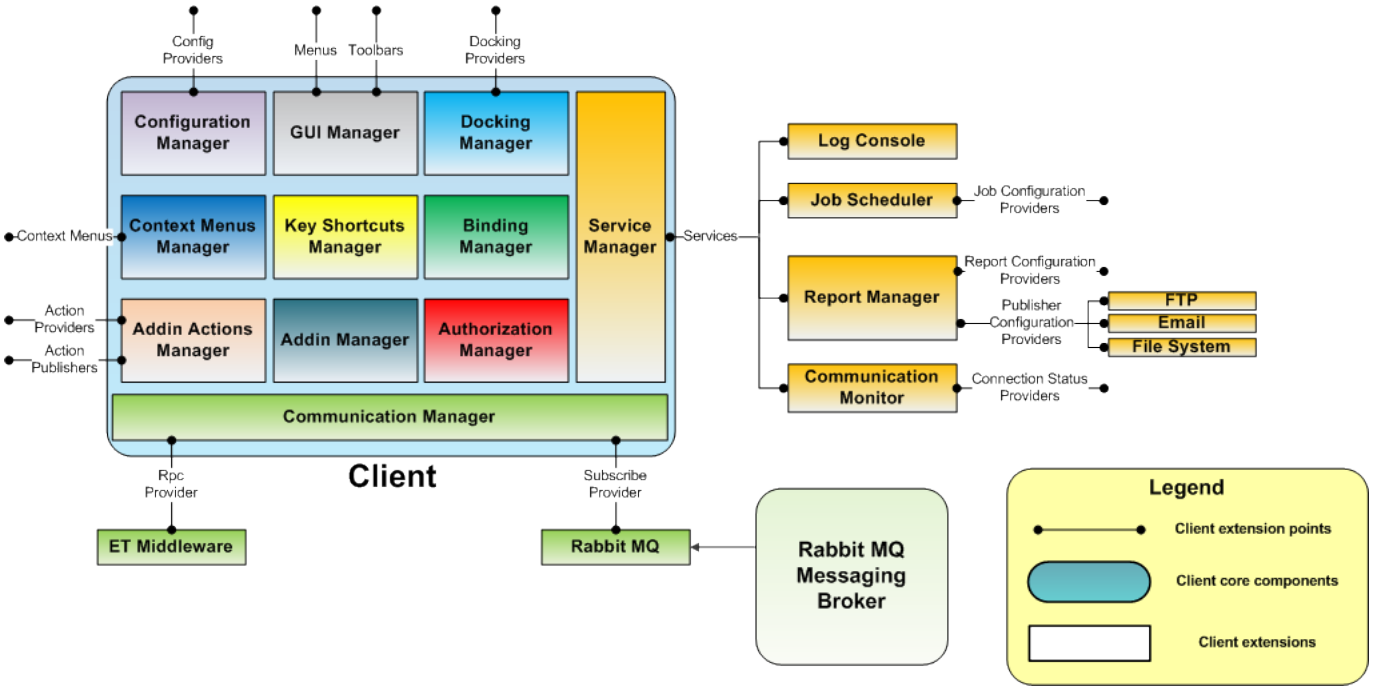


Figure 7.4: Application framework client architecture

Similar to server side, there are couple of core components, some of which expose extension points, so they can be extended by external modules in the future and few of them are already present in the current imple-

mentation. All client side libraries and classes are part of `WAC.ET.Framework.Client` namespace and the core namespace, which encapsulates all core client components, is `WAC.ET.Framework.Client.Core`. The first important class initialized after the client start-up is a `ServiceManager` responsible for loading and initializing all external client modules. These modules have to extend a *Services* extension point of core library, which is declared in a same way as on server side and is described in Section 7.2. The only difference is extension point's path, which is */Client/Services*. Each external module then has to implement an `IComponent` interface also already described in Example 7.2.

Most of the core client components will be described in separate sections, which follow this paragraph except two of them that do not need a detailed description. The first is an `AuthorizationManager`, that is used for client authorization and authentication and for caching the user's permissions on client side and the second one is an `AddinManager`, which provides a dialog for displaying currently loaded modules and in the future will allow installing them from online or local repositories and enabling or disabling them at runtime.

7.3.1 Communication

Communication layer on client side is technically implemented in a similar way as on server side. It exposes two extension points, which declaration is shown in Example 7.10. It is an extension point for RPC communication provider, that is loaded through `RpcCommunicationProviderExtensionNode` and an extension point for subscription communication provider loaded through `SubscriptionCommunicationProviderExtensionNode`. Both extension nodes can be described with same attributes as communication extension nodes on server side, so for their description see Section 7.2.1.

Example 7.10 Client side communication extension points declaration

```
<ExtensionPoint path = "/Client/Communication/RpcProvider">
  <ExtensionNode name="RpcCommunicationProvider"
    type="WAC.ET.Framework.Client.Core.Communication.RpcCommunicationProviderExtensionNode">
    <Description>
    Registers a RPC communication provider allowing request-response communication with server.
    </Description>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Client/Communication/SubscriptionProvider">
  <ExtensionNode name="SubscriptionCommunicationProvider"
    type="WAC.ET.Framework.Client.Core.Communication.SubscriptionCommunicationProviderExtensionNode">
    <Description>
    Registers a subscription communication provider allowing client to subscribe for real-time
    data published by server.
    </Description>
  </ExtensionNode>
</ExtensionPoint>
```

Both communication providers have to implement a corresponding interfaces, which are described in Example 7.11 and Example 7.12.

Example 7.11 IRpcCommunicationProvider interface declaration

```
public interface IRpcCommunicationProvider : IComponent
{
  /// <summary>Gets information about the properties of currently used communication channel.</summary>
  /// <value>Information describing the properties of currently used communication channel.</value>
  ClientRpcInfo RpcInfo { get; }

  /// <summary>Checks if the user is authorized to use the application.</summary>
  /// <param name="request">Logon request carrying information about the user and currently used
  ///                       underlying communication channels.</param>
  /// <returns>Logon respone indicating if the communication with server was successfully
  ///          initialized and if the user is allowed to login into application.
  /// </returns>
  LogonResponse Authorize(LogonRequest request);

  /// <summary>Cancels the request with given requestId.</summary>
  /// <param name="requestId">Identifier of the request, which should be canceled.</param>
  /// <returns><c>true</c> if the cancellation succeeds, <c>false</c> if it fails.</returns>
  bool CancelRequest(int requestId);

  /// <summary>Sends one way request.</summary>
  /// <param name="request">Request to be sent.</param>
  /// <returns>Identifier of currently sent request, which can be used for its cancellation.</returns>
  int SendOneWayRequest(RpcRequest request);

  /// <summary>
  ///  Sends request in asynchronous way and the result of the request is returned through
  ///  registered callback.
  /// </summary>
  /// <typeparam name="T">Type of response returned in a registered callback.</typeparam>
  /// <param name="request">Request to be sent.</param>
  /// <param name="callback">Callback, which should be called when the response for
  ///                        currently sent request is received.</param>
  /// <returns>Identifier of currently sent request, which can be used for its cancellation.</returns>
  int SendRequest<T>(RpcRequest request, RpcRequestCallback<T> callback) where T : RpcResponse;

  /// <summary>Sends request and waits synchronously for its response.</summary>
  /// <typeparam name="T">Type of returned response.</typeparam>
  /// <param name="request">Request to be sent.</param>
  /// <returns>Response to currently sent request.</returns>
  T SendRequestAndWaitForResponse<T>(RpcRequest request) where T : RpcResponse;

  /// <summary>Starts the RPC channel's communication.</summary>
  /// <param name="connectionId">Channel identification.</param>
  void StartRpcCommunication(string connectionId);
}
```

Example 7.12 ISubscriptionCommunicationProvider interface declaration

```
public interface ISubscriptionCommunicationProvider : IComponent
{
  /// <summary>Gets information about the properties of currently used communication channel.</summary>
  /// <value>Information about the properties of currently used communication channel.</value>
  ClientPublishInfo PublishInfo { get; }

  /// <summary>Tells the provider to start subscription for server event messages.</summary>
  /// <param name="manager">EventManager to which should be passed all received event messages.</param>
  void SubscribeForEvents(EventManager manager);

  /// <summary>Tells the provider to start subscription for server log messages.</summary>
  /// <param name="manager">LogManager to which should be passed all received log messages.</param>
  void SubscribeForLog(LogManager manager);
}
```

The class responsible for loading and initializing both communication providers is `CommunicationMa-`

nager. Besides the discovery and instantiation of communication extensions it is used by all other client components as a single point for communication. It provides interface methods from both communication providers and passes all communication requests to them. It also contains couple of common exception handlers, which are called whenever some exception occurs during the communication process, so the communication failures don't have to be handled in individual components.

The most significant difference between client and server side communication layer implementation is that on server side, each extension point can load unlimited number of communication providers, whilst on client side, there can be only one communication provider for each extension point. In Section 7.2.1 it was mentioned, that current implementation of application framework server includes an electronic trading middleware extension for RPC communication and RabbitMQ middleware extension for publish-subscribe communication channel, so it is quite clear, that the same applies for client, so the communication between the server and client is possible.

7.3.2 Configurations

All important classes responsible for application configuration management can be found in `WAC.ET.Framework.Client.Core` library under the `WAC.ET.Framework.Client.Core.Configuration` namespace. The first class initialized from this namespace is `ConfigurationManager`, which is used for caching the application configurations on client side and providing them to interested client components. It also listens for all configuration changes on server side, so whenever there is a new configuration or an existing one is updated or deleted, the `ConfigurationManager` receives the update event, updates its cache accordingly and distributes the changes to interested components within the client modules.

Because the `ConfigurationManager` doesn't know, what external modules will be loaded during the application start-up, it exposes an extension point through which each client component can register a new `IConfigurationProvider` instance and a set of supported configuration types. As you can see in Example 7.13, two extension nodes are used for this registration, the `ConfigurationProviderExtensionNode`, which loads and initializes an underlying `IConfigurationProvider` instance and a child node of type `ConfigurationExtensionNode`, which tells the `ConfigurationManager`, what configuration types the loaded `IConfigurationProvider` instance supports.

Example 7.13 Configuration providers extension point declaration

```
<ExtensionNodeSet id="ConfigurationProviderNodeSet">
  <ExtensionNode name="ConfigurationProvider"
    type="WAC.ET.Framework.Client.Core.Configuration.ConfigurationProviderExtensionNode">
    <ExtensionNode name="Configuration"
    type="WAC.ET.Framework.Client.Core.Configuration.ConfigurationExtensionNode" />
  </ExtensionNode>
</ExtensionNodeSet>

<ExtensionPoint path = "/Client/Configuration/Providers">
  <ExtensionNodeSet id="ConfigurationProviderNodeSet" />
  <Description>
    Registers a new configuration provider with a list supported of application configurations.
  </Description>
</ExtensionPoint>
```

The `ConfigurationProviderExtensionNode` contains only one manifest attribute, a `type`, which tells the extension node of what type is the underlying configuration provider, so it can be dynamically in-

stantiated and initialized. The `ConfigurationExtensionNode` has besides the `type` attribute also `displayInGeneralSettings` and `generalSettingsPath` attributes. These two attributes tell the `ConfigurationManager`, whether the configuration should be displayed in application's general options and if so, what is its path in general options hierarchy tree.

The only functionality, that each configuration provider has to provide, is the ability to create a configuration instance based on the given type and content. Therefore the declaration of `IConfigurationProvider` interface is very simple and is shown in Example 7.14.

Example 7.14 IConfigurationProvider interface declaration

```
public interface IConfigurationProvider
{
  /// <summary>Indicates whether the provider is initialized or not.</summary>
  bool IsInitialized { get; }

  /// <summary>Creates a configuration based on a given configuration details.</summary>
  /// <param name="configuration">Configuration details.</param>
  /// <returns>Created configuration.</returns>
  BaseConfiguration CreateConfiguration(ConfigurationDetail configuration);

  /// <summary>Initializes the provider.</summary>
  void Initialize();

  /// <summary>Uninitialize the provider.</summary>
  void Uninitialize();
}
```

As you can see, the `CreateConfiguration` method returns an instance of `BaseConfiguration` class, that is the abstract class, from which all application configurations have to derive. It encapsulates the basic properties of the configuration such as its name and description, flag indicating whether the configuration is a default one for given type, the user to whom the configuration belongs and few others. Besides the basic properties it defines also few abstract methods, which each configuration has to implement. These are methods for configuration's serialization, deserialization and cloning. In case the configuration supports the modifications of its properties through user interface, it has to provide an instance of a configuration panel, which will be used for such user interactive modifications. Each configuration panel have to derive from `BaseConfigurationPanel` abstract class, which is then used by other GUI components and dialogs such as dialog for configuration management, dialog for saving a configuration, general application options dialog and few others.

7.3.3 User interface

The most important requirements about user interface implementation were its common look & feel and its flexibility. To provide a common look & feel of application, it was decided to use a third-party open source library called Krypton Toolkit. Krypton Toolkit is a free set of user interface controls and components and is a part of licensed Krypton Suite package provided by Component Factory Pty Ltd [CompFact10]. Besides it provides many useful, feature rich and great looking controls, it also includes a palette system, which allows the user to switch between visual themes at run-time. The library is extensively documented and provides many examples covering all controls.

To give application's user interface as much flexibility as possible a docking framework was used for its implementation. The best open source docking framework for .NET Windows Forms is definitely a DockPanel Suite [DockPanel10] initially developed by Weifen Luo. The library mimics a Visual Studio .Net

docking system, which provides functionality to drag the application windows to custom positions, dock them together under several tabs, hide the windows to sidebars and many other useful features to allow the user to create, save and load complex window hierarchies. To get a better picture, what functionality such docking library provides, please see [Postulka10] included on the attached CD.

To simplify a development of client side modules and to allow their easy integration to application's user interface, several components, controls and classes were implemented under a `WAC.ET.Framework.Client.Core.Gui` namespace and the most important ones will be described in the following paragraphs.

It is a common scenario, that client application has a one main menu and one or several toolbars. Application framework client sticks to this scenario. Component responsible for creation of menu and toolbars is named `MenusAndToolbarsManager`. To allow dynamically loaded external modules to integrate their functionality to application menu and toolbars, two extension points are exposed by core library and their declaration is shown in Example 7.15.

Example 7.15 Main menu and toolbars extension points

```
<ExtensionNodeSet id="MenuNodeSet">
  <ExtensionNode name="MenuItem" type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.MenuNodeItem"/ ↩
      >
  <ExtensionNode name="MenuSeparator"
                 type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.MenuNodeSeparator" />
  <ExtensionNode name="Menu" type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.SubmenuNode">
    <ExtensionNodeSet id="MenuNodeSet" />
  </ExtensionNode>
</ExtensionNodeSet>

<ExtensionNodeSet id="ToolbarNodeSet">
  <ExtensionNode name="Toolbar" type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.ToolbarNode">
    <ExtensionNode name="ToolItem"
                   type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.ToolNodeItem" />
    <ExtensionNode name="ToolSeparator"
                   type="WAC.ET.Framework.Client.Core.Gui.MenusAndToolbars.ToolNodeSeparator" />
  </ExtensionNode>
</ExtensionNodeSet>

<ExtensionPoint id="MainMenu" path="/Client/Gui/MainMenu">
  <Description>Allows to register new menu items to main application menu.</Description>
  <ExtensionNodeSet id="MenuNodeSet" />
</ExtensionPoint>

<ExtensionPoint path = "/Client/Gui/Toolbars">
  <Description>Allows to register new toolbars and toolbar items.</Description>
  <ExtensionNodeSet id="ToolbarNodeSet" />
</ExtensionPoint>
```

As you can see, there are several extension nodes for different types of menu and toolbar items defined. What should be noted here is that menu node set is defined recursively, so the modules can easily extend sub menus as well and create a complex menu hierarchy. Each menu or toolbar item extension node, except the separators, contains `icon`, `text` and `commandType` attributes telling the `MenusAndToolbarsManager`, what icon and text should be displayed for given item and what `ICommand` should be used for it. `ICommand` interface declares only one `Run` method, which is executed whenever the user clicks on menu or toolbar item with the corresponding command. Each command can be used for several items, so there is no need to define different commands for menu and toolbar items if they are providing the same functionality. To tell `MenusAndToolbarsManager`, in which order should be these items added to menu or toolbars, developer can use built-in Mono.Addins `insertbefore` and `insertafter` attributes. These two attributes are used to specify the relative location of a node within add-in hierarchy, so

when add-in engine is passing the extension nodes on a given extension point to an application, it ensures that if `insertafter`/`insertbefore` is specified, the node will be passed to the application after/before the specified node, and before/after any other node defined in the same extension.

Similar to application menu and toolbars, it was required to allow an easy integration of windows created by external modules into a Weifen Luo's docking framework. Therefore a couple of helper classes and controls were implemented and can be found in `WAC.ET.Framework.Client.Core.Gui.Docking` namespace. To centralise a docking functionality into one place, a `DockingManager` was implemented. It is the core docking system class and is initialized during the application start-up. It handles the application layout creation process, contains the methods to save a current layout and load it at a later time, contains a reference to a main `DockPanel` instance, which is used to open and dock new docking window instances and subscribes for layout update events, so whenever a new layout is created or an existing one is updated or deleted, it receives from the application framework server an event about these changes and propagates this event to the application interface. The `DockingManager` is also responsible for dynamic discovery, loading and initializing of `IDockingProvider` instances from external modules.

Each external module bound into service extension point, which wants to provide docking functionality on its windows, have to register an `IDockingProvider` instance. Such registration is provided through an extension point declared in Example 7.16 and exposed by `WAC.ET.Framework.Client.Core` library.

Example 7.16 Docking providers extension point

```
<ExtensionPoint path = "/Client/Gui/DockingProviders">
  <ExtensionNode name="DockingProvider"
    type="WAC.ET.Framework.Client.Core.Gui.Docking.DockingProviderExtensionNode">
    <Description>Registers a new docking provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

The only important attribute of `DockingProviderExtensionNode` is a `type` attribute telling the extension node of what type an underlying `IDockingProvider` instance is, so it can be dynamically instantiated. The `IDockingProvider` interface, which declaration is shown in Example 7.17, declares besides the common initialization methods, two important methods, which have to be implemented by each registered provider instance. Both methods are used by a `DockingManager` whenever it loads a new application layout. The `GetAllContents` method is used to retrieve all currently open windows by the given provider, so common uninitialization methods can be called on each currently open docking window before the current application layout is disposed. The `GetContent` method is called afterwards and is used to retrieve a correct instance of persisted window in layout structure, which should be loaded. Why it is necessary to instantiate a window through a corresponding `IDockingProvider` instance is quite clear, the `DockingManager` doesn't know the window types from external libraries and therefore have to pass the process of window creation to corresponding docking providers.

Example 7.17 IDockingProvider interface declaration

```
public interface IDockingProvider
{
  /// <summary>Indicates whether the provider is initialized or not.</summary>
  bool IsInitialized { get; }

  /// <summary>Gets the name of the provider.</summary>
  string Name { get; }

  /// <summary>Gets all currently open docking windows.</summary>
  List<DockContent> GetAllContents();

  /// <summary>Gets an instance of window of a given type.</summary>
  /// <param name="typeString">      The type of the window, which should be created.</param>
  /// <param name="customAttributes">The custom windows creation attributes.</param>
  /// <returns>Instance of created window.</returns>
  IDockContent GetContent(string typeString, string customAttributes);

  /// <summary>Initializes the provider.</summary>
  void Initialize();

  /// <summary>Uninitialize the provider.</summary>
  void Uninitialize();
}
```

The last rule related to a docking system is, that each application window, which should benefit from docking features have to derive from a `DockingWindow` class. This class was created to centralise a common functionality of each window into a one place, so the external modules don't have to re-implement this functionality with each newly added window. There are a couple of virtual properties and methods defined on a `DockingWindow` class, which can each derived window override and customize the behaviour of window instance a little bit. As an example, programmer can tell the window if the window configuration is supported or not and if so, the base class will automatically adjust the window context menu and allow the user to create new configurations for a given type of window and to choose between them in the future. Programmer can specify a list of supported window tab parameters, which can be then used when the user wants to rename a window tab. Programmer can also override a supported list of events, about which the window wants to be notified etc. Besides this customization functionality, the `DockingWindow` class contains an implementation of serialization process, which is used for persisting the important information about the window whenever the application layout is saved by the `DockingManager`, so the window can be recreated at a later time.

7.3.4 Module interaction

Because application framework architecture is developed as a plug-in based system, it has to take into account, that many new modules will be integrated into framework in the future. It is a common scenario, that these modules can function fully independently, but at the same time, they could provide some functionality, which could be interesting for other modules. As an example of such interaction in monitoring system can be provided the message detail view, which reacts on a focus change in a message hierarchy window. To allow sharing functionality between different modules and provide a module interaction, couple of components and classes were developed and integrated into application framework client.

The first important step was to develop some type of notification system, which will allow sending messages between independent modules. The biggest advantage of such messaging system against the usage of direct calls on the interfaces that they support and implement is, that the external modules become very loosely

coupled. The components involved in the interaction process don't need to know any details about the component they communicate. The only shared object is a version of the messages being sent, not the type or version of the components that process them. After some research, it was decided, that it is not needed to implement such messaging system, because a free notification library already exists and therefore it was used for application framework implementation. This free library is called Acrux Threading .NET [Acrux10] and is developed by Acrux Software, an ISV member of the Microsoft Partner program. It provides a subscription based notification system that allows an interaction between components within the same process, so one can easily integrate complex multithreaded sub-systems in a loosely coupled manner.

Even though, the Acrux Threading.NET library provides a complex notification system, a couple of other components and structures were implemented, so the notification system could be easily integrated with other user interface components. All basic classes, which extend the notification system, are implemented in `WAC.ET.Framework.Client.Core.Actions` namespace. To better understand what role these classes play in the notification architecture please see the Figure 7.5.

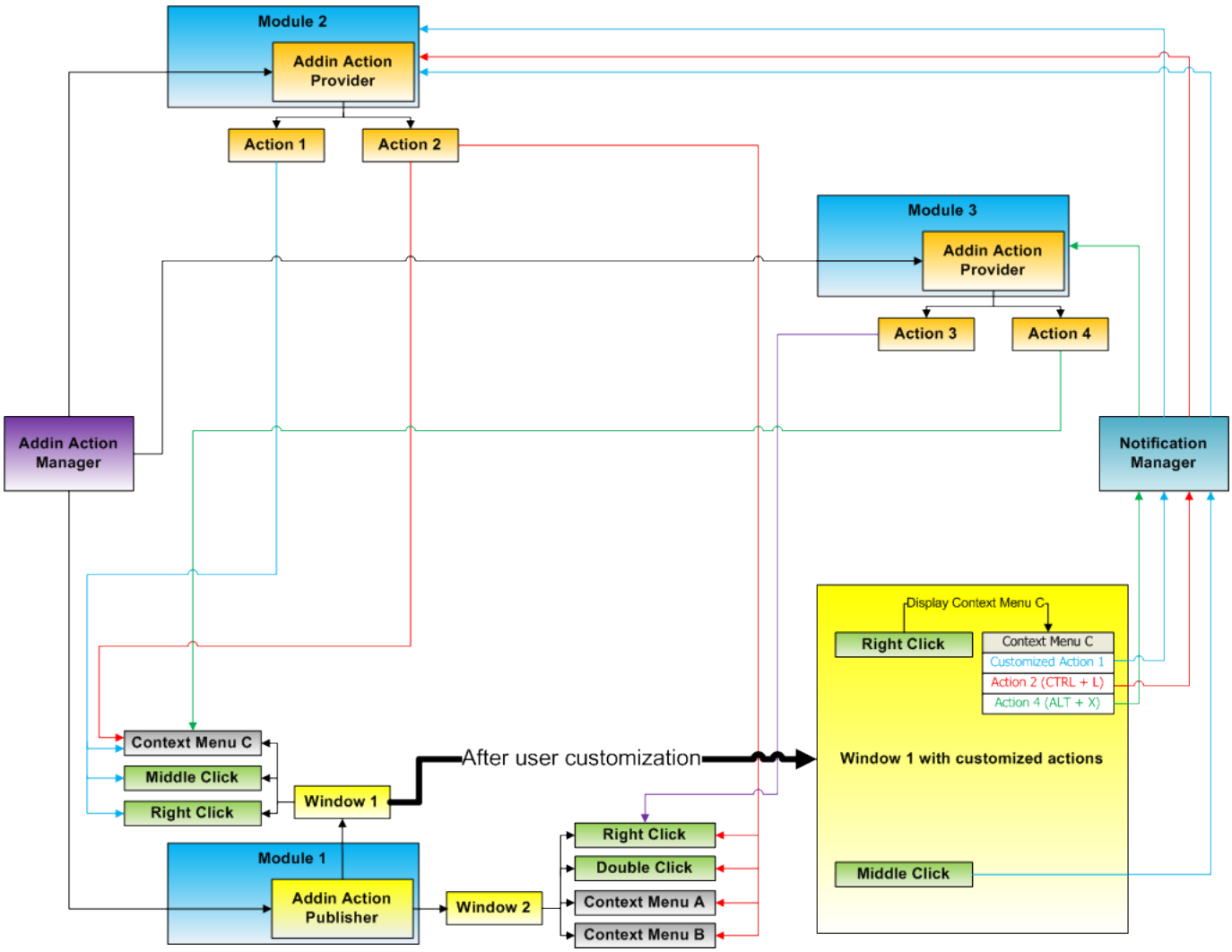


Figure 7.5: Notification system and module interaction

As you can see, there are three application modules. The first module registers an `AddinActionPubl-`

isher instance, which provides two different windows. Each window then defines a set of custom events, as an example different types of mouse clicks, and a set of context menus. Two other modules register an AddinActionProvider instance, which is used to tell the AddinActionManager, what actions the given module provides and from which modules these actions can be executed. Each action, represented by AddinAction class, can be connected to several different types of custom events and context menus defined by AddinActionPublisher instance. In the architecture example above, the Action 2 can be executed from all context menus or by all custom events defined on Window 2 or it can be executed from Context Menu C on Window 1. What should be noted here is, that to each context menu or custom event can be connected several different actions. The application user then can customize, which action on which event should be executed, which actions and under what names and with what order and hierarchy will be displayed in published context menus and what keyboard shortcuts should be assigned to each action. An example of such customization is illustrated in figure above on Window 1. The user have chosen, that on this window he wants to display a Context Menu C whenever he clicks the right mouse button and that the Action 1 should be executed whenever he clicks the middle mouse button. It was also defined, that Context Menu C should contain all three connected actions, where first action is displayed with customized text. The last customization is related to keyboard shortcuts, where user has defined, that Action 2 will be executed with CTRL+L key combination and Action 4 with ALT+X key combination. Notification, that some action should be executed is then passed to a corresponding module through a NotificationManager instance, which is a part of already mentioned notification system from Acrux Threading.NET library. Once the correct module receives the notification, it takes the execution parameters, which were passed from publishing module and executes the action accordingly.

The example above is a very simple scenario chosen to illustrate the main purpose of such notification system, but it is quite clear, that each module can define tens of different actions, windows, context menus and custom events and that with such flexibility and ease of registration, the creation of complex module interaction hierarchy is a very simple process. The only thing, the developer of a new module have to define is, what actions can be executed by this module and which already existing modules should be extended by these actions and in which way, if it should be through context menus, custom events or both. It is also quite clear, that newly added actions will be visible to user only in case the module is loaded, and if it is not, the user will not see them and can't use them during the customization process. To see, how this is used in a monitoring system, please see the [Postulka10] included on the attached CD.

Now, when you have a picture about how the notification process works, a few technical details will be shortly described, so the developer of new module knows, how to achieve such integration. The first important and already mentioned class is an AddinActionManager. It is responsible for discovering all AddinActionPublisher and AddinActionProvider instances. During the discovery process, it builds the complete hierarchy and stores it into memory, so it can be used by other helper classes like ContextMenusManager responsible for context menu customization process and ShortcutsManager, which provides functionality to customize application keyboard shortcuts. The registration process of new AddinActionPublisher and AddinActionProvider instances is implemented through corresponding extension points exposed by WAC.ET.Framework.Client.Core library and their declarations are shown in Example 7.18.

Example 7.18 Add-in action extension points declaration

```
<ExtensionPoint path = "/Client/Actions/Providers">
  <ExtensionNode name="ActionProvider"
    type="WAC.ET.Framework.Client.Core.Actions.AddinActionProviderExtensionNode">
    <ExtensionNode name="Action" type="WAC.ET.Framework.Client.Core.Actions.AddinActionExtensionNode" >
      <ExtensionNode name="PublisherRef"
        type="WAC.ET.Framework.Client.Core.Actions.AddinActionPublisherRefExtensionNode" >
        <ExtensionNode name="ContextMenuRef"
        type="WAC.ET.Framework.Client.Core.Actions.AddinActionContextMenuRefNode" />
        <ExtensionNode name="CustomEventRef"
        type="WAC.ET.Framework.Client.Core.Actions.AddinActionCustomEventRefNode" />
      </ExtensionNode>
    </ExtensionNode>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Client/Actions/Publishers">
  <ExtensionNode name="ActionPublisher"
    type="WAC.ET.Framework.Client.Core.Actions.AddinActionPublisherExtensionNode">
    <ExtensionNode name="Window"
      type="WAC.ET.Framework.Client.Core.Actions.AddinActionPublisherWindowExtensionNode">
      <ExtensionNode name="ContextMenu"
      type="WAC.ET.Framework.Client.Core.Actions.AddinActionPublisherContextMenuExtensionNode" />
      <ExtensionNode name="CustomEvent"
      type="WAC.ET.Framework.Client.Core.Actions.AddinActionPublisherCustomEventExtensionNode" />
    </ExtensionNode>
  </ExtensionNode>
</ExtensionPoint>
```

As you can see, the hierarchy of extension nodes corresponds to the hierarchy illustrated in Figure 7.5. Each extension node declares a couple of attributes, which allow the module to describe the extension node.

The last important part of module interaction process is so called binding mechanism between several application windows. This mechanism is also developed as an extension to an Acrux Threading.NET library and its main purpose is to provide the developers an easy way, how to create a type of master-detail views between application windows. Its implementation is very similar to previously described notification process. The main difference is, that the binding is much more dynamic then customization of context menus, keyboard shortcuts and custom events. Therefore specialized components under `WAC.ET.Framework.Client.Core.Gui.Binding` namespace were implemented. It is a `BindingManager` class, which manages the windows binding hierarchy, is responsible for dynamic subscriptions and unsubscriptions and persists binding definitions during the saving process of application layout and reinitializes them when the layout is loaded at later time.

To centralise the binding functionality of application windows to one place, the namespace also provides three window controls derived from base `DockingWindow` class. It is a `BindingParentWindow`, which represents a master view, the `BindingChildWindow` representing a detail view and a `BindingWindow`, which mimics the both of them. Each window, which wants to be a part of a binding mechanism have to derive from one of these three classes according as what role of binding mechanism it wants to play. Each derived parent window then have to specify a `BindingPublisher`, which tells the `BindingManager`, to what types of events child windows can subscribe and each derived child window have to specify, to what types of `BindingPublishers` and what types of events it wants to be able to subscribe. This subscription is done by overriding a corresponding methods on base class. To get a better picture how the binding mechanism works, please see the [Postulka10] included on the attached CD.

7.3.5 Built-in extensions

The application framework client contains besides the built-in communication providers already mentioned in a Section 7.3.1 a couple of other useful extensions, which provide a common functionality to all client components.

The first is a `CommunicationMonitor` module developed as a service extension, which provides an easy way how to monitor the connection status of all communication channels used by client modules. Each module, which uses some type of communication can extend the `CommunicationMonitor` module through a connection monitor extension point shown in Example 7.19.

Example 7.19 Communication monitor extension point declaration

```
<ExtensionPoint path = "/Client/ConnectionMonitor">
  <ExtensionNode name="ConnectionStatusProvider"
    type="WAC.ET.Framework.Client.Communication.Monitor.ConnectionMonitorExtensionNode">
    <Description>Registers a new connection status provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

The only attribute declared on `ConnectionMonitorExtensionNode` is a type, through which the module have to specify of what type is an underlying `IConnectionStatusProvider` instance, so it can be dynamically instantiated. The `IConnectionStatusProvider` interface, which declaration is shown in Example 7.20, contains few properties and an event handler, which should be raised whenever the status of communication channel has changed. Interface properties are then used to define a module's logical path and a list of supported connection states, which could be displayed in a connection monitor window. Both electronic trading and RabbitMQ communication providers extend the `Communication-Monitor` module and provide the connection status of all their communication channels.

Example 7.20 IConnectionStatusProvider interface declaration

```
public interface IConnectionStatusProvider
{
  /// <summary>Gets type of enumeration which represents set of possible connection statuses.</summary>
  Type EnumType { get; }

  /// <summary>Gets id of the provider.</summary>
  string Id { get; }

  /// <summary>Gets logical path to systems for which is the connection status provided.</summary>
  string Path { get; }

  /// <summary>Raised whenever the connection status of communication channel changes.</summary>
  event ConnectionStatusChangedEventHandler ConnectionStatusChangedEvent;
}
```

The second very useful extension is a `LogConsole`, which is developed as a simple window displaying all application log events. It subscribes to a centralised `LogStorage` instance, which is derived from `Text-Writer` class, so it can be set as the output of `Application.Console` instance. This provides a very easy way how to visualize all application log events, which are logged through third-party log4net library. The other advantage is that the log level can be dynamically changed at runtime. Besides the listening to all log4net messages, the `LogStorage` instance also listens to a server side log events and pass them to a log console window, so user can see what is happening on server side directly in a client terminal. If

some module can't or do not want to use a log4net library for logging functionality, it can directly call an `AddLogMessage` method on `LogStorage` instance and specify what message and under which group the log console should display. Each log group is then visualized in a log console window as a separate tab control.

Last two built-in client service extensions are `JobScheduler` and `ReportManager` modules. These two modules are client side implementation of corresponding server side modules already mentioned and described in Section 7.2.2 and Section 7.2.3. These modules provide a user interface and functions to control the server modules and expose the extension points for client side configuration providers used to configure and visualize jobs, reports and report publishers. Because declarations of extension points and configuration provider interfaces are very simple, there is no need to discuss all of them here and for details please see [Postulka10a].

Chapter 8

Monitoring System

As it was already mentioned in previous chapter, the monitoring system was developed as a set of external modules using an application framework. To get a better picture about monitoring system and to better understand the following sections, its integration to application framework and electronic trading infrastructure is illustrated in Figure 8.1.

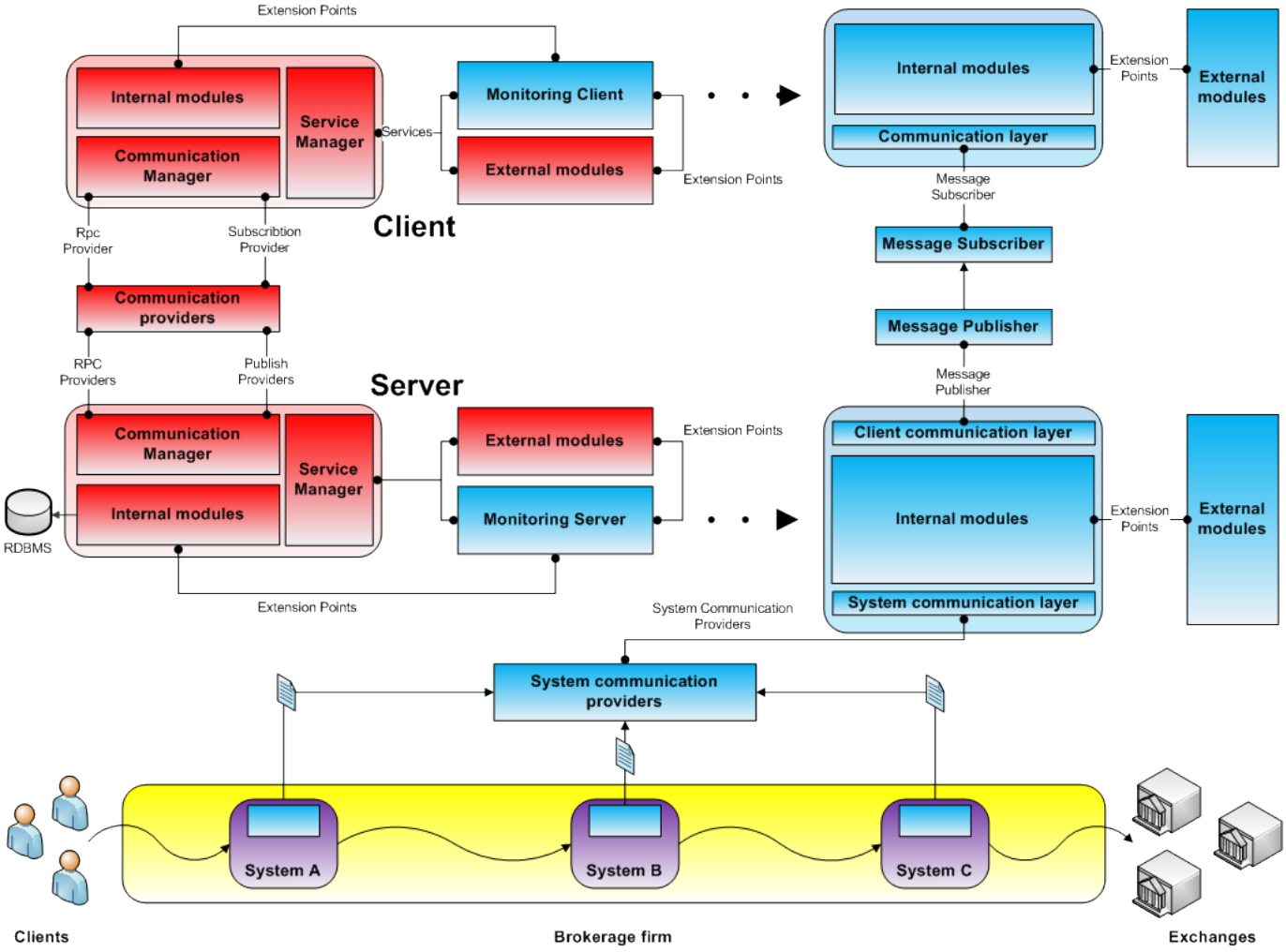


Figure 8.1: Monitoring system integration

As you can see, the monitoring system implementation is split into a server and client parts the similar way as an application framework itself and both are then implemented as service extensions, which extend the internal and external application framework modules too. The server part is responsible for monitoring the electronic trading infrastructure, processing the received messages and publishing them to monitoring clients, which visualize the message flow to the end user. Both modules contain several internal and external components and their usage and implementation will be separately described in the following sections.

8.1 Server implementation

The main reason to implement a monitoring server was to centralise a message processing. Server is responsible for communication with electronic trading systems, for message identification and additional data calculation and for distribution of messages to monitoring clients. It is also responsible for persistence of all monitoring structures, and then providing them to clients whenever user requests them. To get an overview about what components and modules comprise monitoring server, its architecture is illustrated in Figure 8.2.

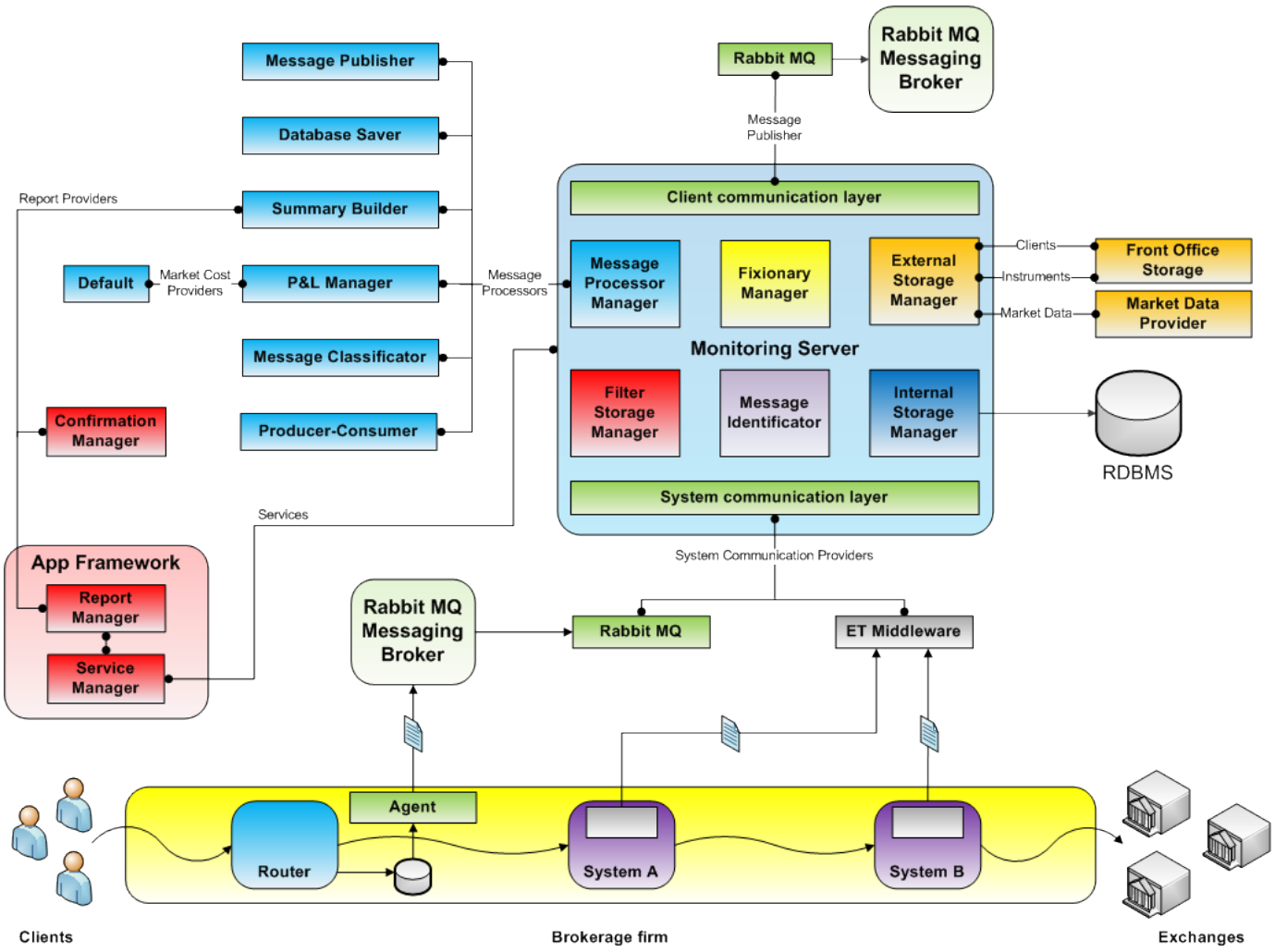


Figure 8.2: Monitoring server architecture

As you can see, there is a couple of internal components, which form a core of monitoring system and some of them expose the extension points, so the monitoring server architecture can be easily extended in the future. The following sections describe the most interesting components and modules, so you can better understand how the monitoring process on server side works.

8.1.1 Internal storage

The major part of important classes responsible for monitoring structure persistence is implemented in `WACET.Monitor.Server.Core.Storage.Internal` namespace. It contains a couple of storage managers, which are described in the following paragraph.

- `MarketStorage` - storage responsible for market management.
- `SystemStorage` - storage containing information about all currently monitored systems, their names, location, currently configured FIX and internal sessions and information about what communication providers should be used for their monitoring.

- `MessageFilterStorageManager` - contains all user and programmatically defined message filters used to filter messages based on a set of rules.
- `MessageFormatStorageManager` - contains all user and programmatically defined message formats, which are used to customize message representation.
- `MessageStyleStorageManager` - contains all user and programmatically defined message styles used to customize message visualization styles.

Another important manager, which should be mentioned in this section, is a `FixionaryManager` used to cache and provide all information about FIX and internal electronic trading middleware protocol. It is very important component, because such information is required for both message processing on server side and message visualization on client side. To get this information, the FIX and internal protocol definitions were parsed and stored into database, so they can be loaded by `FixionaryManager` instance. The parsed data contain details and descriptions about all protocol versions, messages, structures, components and tags and are used throughout the whole monitoring system.

Besides the above mentioned managers, which are responsible for updating, caching and providing user defined or static monitoring structures, it is important to make all messages, received from monitored systems, persistent, so they can be retrieved and processed in the future. The database persistence of messages is fundamental for monitoring system history support. Such a feature is not implemented completely at the moment, because it doesn't belong to the must-to-have requirements, but the system should be prepared for its future support. Therefore the functionality of storing messages and their additional data within the database is already present in current solution and the only thing, which have to be added to fully support the historical views is a set of dynamic queries, which will return the requested messages in a requested format, so the history view can be maintained with same application processes as a real-time one.

Because it is expected, that the monitoring system will be processing millions messages per day, the partitioning technique was used for a largest system tables. This technique was introduced with MS SQL server 2005 and provides a way to divide large tables and indexes into smaller parts by segregating the data using a particular scheme or set of rules. The main reason for doing this is a query performance. If you partition data with good partitioning scheme, which is relevant to your data, then whenever you query a data and your query sticks to the partitioning scheme rules, it only touches the data in relevant partition. This behaviour is exactly what is needed for monitoring system history support. The most common scenario using the history support is that a user wants to see a set of data from some previous trading session or from a range of few days. Therefore the created partitioning schema is based on a message date, so the largest tables are split into partitions, which correspond to each trading session. The biggest advantage of this partitioning schema is that historical queries can't influence performance of real-time data insertion and that each historical query is executed on data from corresponding date range only, so the queries are much faster. Without this partitioning technique both the history support and real-time message processing would suffer from very slow performance.

8.1.2 External storage

In previous section were described the most important parts of internal storage implementation. Because one of the requirements of monitoring system was to take advantage of data from some already existing systems, an additional external storage was implemented. It is implemented as an extensible component and as illustrated in Figure 8.2 it is used to register external storages, which provide the monitoring system

with client and instrument definitions and also with market data. The main advantages to implement each storage as an external module and not an internal component are the following:

- **Ability to use already existing systems** - Monitoring system can use data from external systems and does not have to duplicate the data itself and the functionality to their management.
- **Ability to easily switch the external storage to a different system** - In case the external system used to provide the data is replaced by a new one, there is no need to change the internal implementation of monitoring system and to deploy a new version. Instead of that, the only thing, which has to be done, is to implement a new external module, which will replace the current one.
- **Data and their management centralisation** - There is no need to maintain already existing data in monitoring system and synchronize them with other systems. Data can be centralised and managed on one place.

All important classes and components are included in a `WAC.ET.Monitor.Server.Core.Storage.External` namespace, which exposes three different extension points through XML add-in manifest of `WAC.ET.Monitor.Server.Core` library. These three extension points are used to register a corresponding external storage and their declaration is shown in Example 8.1.

Example 8.1 External storage extension points

```
<ExtensionPoint path = "/Monitor/Storage/External/Clients">
  <ExtensionNode name="ClientStorage"
    type="WAC.ET.Monitor.Server.Core.Storage.External.ClientStorageExtensionNode">
    <Description>Registers a new client storage.</Description>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Monitor/Storage/External/Instruments">
  <ExtensionNode name="InstrumentStorage"
    type="WAC.ET.Monitor.Server.Core.Storage.External.InstrumentStorageExtensionNode">
    <Description>Registers a new instrument storage.</Description>
  </ExtensionNode>
</ExtensionPoint>

<ExtensionPoint path = "/Monitor/Storage/External/MarketData">
  <ExtensionNode name="MarketDataStorage"
    type="WAC.ET.Monitor.Server.Core.Storage.External.MarketDataStorageExtensionNode">
    <Description>Registers a new market data storage.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

All extension nodes declare the same attributes, a `type` attribute used to instantiate an underlying storage instance and a `configFileName` attribute to specify a configuration, which should be used to initialize the storage. The core class used to discover and initialize all external storages is an `ExternalStorageManager` and is used by all server components whenever they want to access currently loaded storages. Each external storage has to implement a corresponding interface, and these are shown in the following examples.

Example 8.2 IExternalStorage interface declaration

```
public interface IExternalStorage
{
  /// <summary>Gets a value indicating whether the storage is initialized.</summary>
  bool IsInitialized { get; }

  /// <summary>Gets the identifier of the storage.</summary>
  string StorageIdentifier { get; }

  /// <summary>Initializes the storage based on given configuration.</summary>
  /// <param name="configuration">Storage configuration.</param>
  void Initialize(string configuration);

  /// <summary>Refreshes storage's data.</summary>
  void Refresh();

  /// <summary>Uninitialize storage.</summary>
  void Uninitialize();
}
```

Example 8.3 IClientStorage interface declaration

```
public interface IClientStorage : IExternalStorage
{
  /// <summary>Gets the list of clients.</summary>
  List<ClientInformation> Clients { get; }

  /// <summary>Gets the unknown client.</summary>
  ClientInformation UnknownClient { get; }

  /// <summary>Tries to get client based on a given name and market mic code.</summary>
  /// <param name="clientName">   Name of the client.</param>
  /// <param name="marketMICCode">The market mic code.</param>
  /// <param name="client">       [out] The requested client.</param>
  /// <returns><c>true</c> if client exists, <c>false</c> if not.</returns>
  bool TryGetClient(string clientName, string marketMICCode, out ClientInformation client);

  /// <summary>Tries to get client based on a given identifier and market mic code.</summary>
  /// <param name="clientId">     Identifier of the client.</param>
  /// <param name="marketMICCode">The market mic code.</param>
  /// <param name="client">       [out] The requested client.</param>
  /// <returns><c>true</c> if client exists, <c>false</c> if not.</returns>
  bool TryGetClient(int clientId, string marketMICCode, out ClientInformation client);

  /// <summary>Tries to get client based on a given identifier.</summary>
  /// <param name="clientId">Identifier of the client.</param>
  /// <param name="client">  [out] The requested client.</param>
  /// <returns><c>true</c> if client exists, <c>false</c> if not.</returns>
  bool TryGetClient(int clientId, out ClientInformation client);
}
```

Example 8.4 IInstrumentStorage interface declaration

```
public interface IInstrumentStorage : IExternalStorage
{
  /// <summary>Gets a list of instruments.</summary>
  List<InstrumentInformation> Instruments { get; }

  /// <summary>Gets the unknown instrument.</summary>
  InstrumentInformation UnknownInstrument { get; }

  /// <summary>Tries to get instrument with given identifier.</summary>
  /// <param name="instrumentId">Identifier of requested instrument.</param>
  /// <param name="instrument">  [out] The requested instrument.</param>
  /// <returns><c>true</c> if instrument exists, <c>false</c> if not.</returns>
  bool TryGetInstrument(int instrumentId, out InstrumentInformation instrument);

  /// <summary>Tries to get instrument identifier based on given type, name and on given market code.
  /// </summary>
  /// <param name="idSource">      The instrument type.</param>
  /// <param name="instrumentName">Name of the instrument.</param>
  /// <param name="marketMICCode"> The market mic code.</param>
  /// <param name="instrumentId">  [out] Identifier of the instrument.</param>
  /// <returns><c>true</c> if instrument exists, <c>false</c> if not.</returns>
  bool TryGetInstrumentId(IdSource idSource, string instrumentName, string marketMICCode,
              out int instrumentId);
}
```

Example 8.5 IMarketDataStorage interface declaration

```
public interface IMarketDataStorage : IExternalStorage
{
  /// <summary>
  ///  Tries to get currency rate.
  /// </summary>
  /// <param name="sourceCurrency">Source currency.</param>
  /// <param name="targetCurrency">Target currency.</param>
  /// <param name="dateTime">      Time according to the rate should be calculated.</param>
  /// <param name="rate">          [out] Requested currency rate.</param>
  /// <returns><c>true</c> if rate exists, <c>false</c> if not.</returns>
  bool TryGetRate(string sourceCurrency, string targetCurrency, DateTime dateTime, out decimal rate);
}
```

Because each external storage retrieves the data from some external system, it is important to allow refreshing the data on both user request and regular basis. For this purpose, the `ExternalStorageJobsProvider` class was implemented. It is an extension to an application framework's `JobScheduler` module described in Section 7.2.2 and provides a list of jobs, which call during their execution a `Refresh` method of corresponding external storage.

8.1.3 Message identification

One of the most important information, which has to be known about each order, is information about what client sent the order and to which market it was designated. Because the rules to identify these two parameters are not predefined or standardized in FIX protocol, they can vary for each client. Therefore it was necessary to implement a component, which will allow the user to create a set of identification rules and which will be responsible for message identification. The implementation of identification process is illustrated in Figure 8.3.

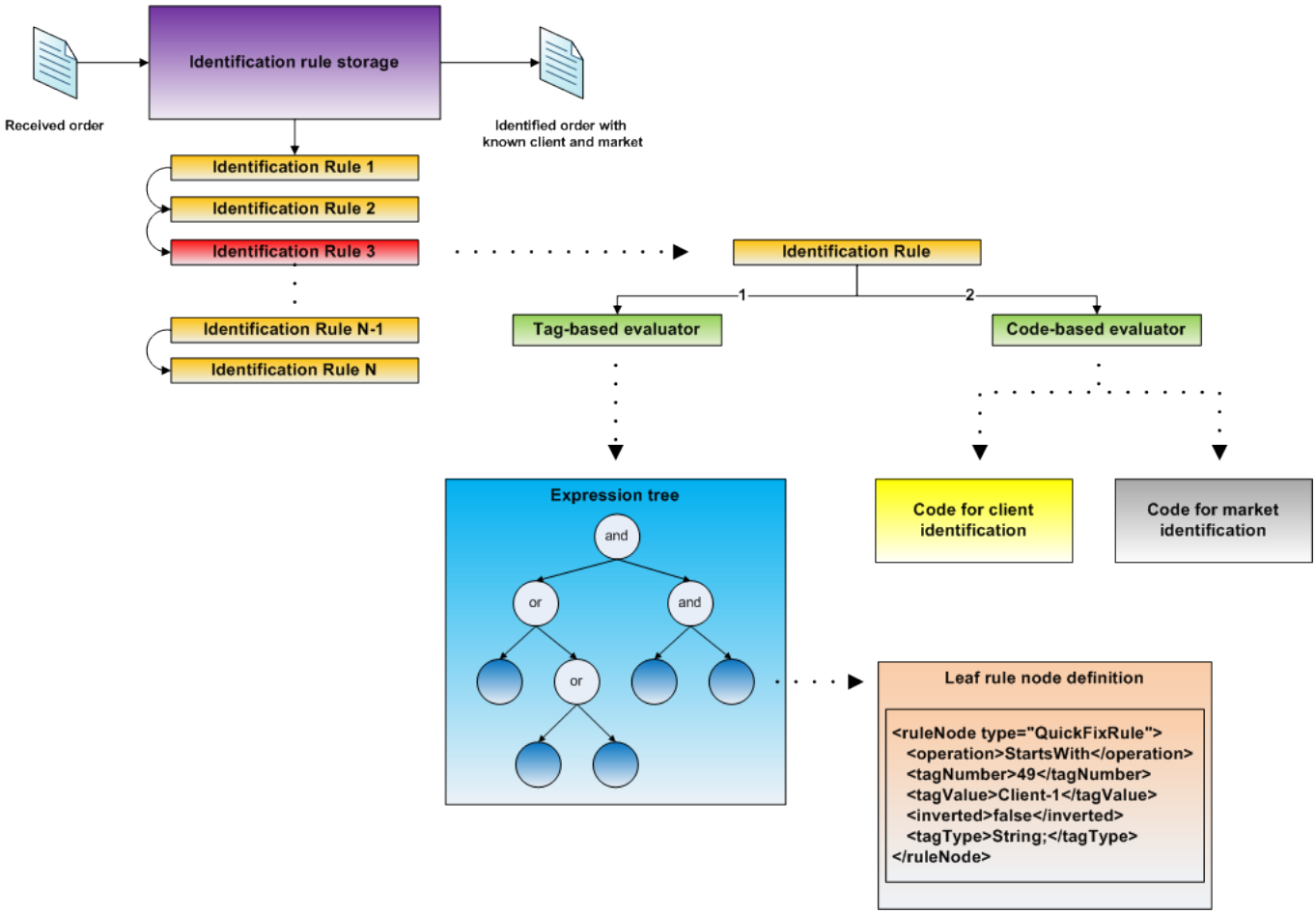


Figure 8.3: Message identification process

As you can see, there is an identification rule storage, which contains all currently available identification rule instances and is responsible for their management. Whenever the monitoring system receives a new order, it passes this order to a storage instance, which tries to match the order to one of the existing rule. As illustrated in a picture above, each identification rule is split into two parts. The first part is a tag-based evaluator used to identify if the incoming order matches the given rule and a second part is a code-based evaluator through which the client and market identifiers are retrieved.

The tag-based evaluator is implemented as an expression tree, where each inner node is a logic operation and each leaf node is a tag-based rule, which defines what tag should be compared with what value and what comparison operation should be used. This allows the user to create an expression tree with a set of tag-based rules put to an expression formula. The whole expression tree is then stored in a database as an xml structure and its example is shown in Example 8.6.

Example 8.6 Expression tree xml structure

```
<evaluator>
  <conjNode type="Or" errorMessage="">
    <conjNode type="And" errorMessage="">
      <ruleNode type="QuickFixRule" errorMessage="">
        <operation>Equal</operation>
        <tagNumber>49</tagNumber>
        <tagValue>CLIENT1</tagValue>
        <inverted>false</inverted>
        <tagType>String;</tagType>
      </ruleNode>
    <ruleNode type="QuickFixRule" errorMessage="">
        <operation>StartsWith</operation>
        <tagNumber>109</tagNumber>
        <tagValue>1900</tagValue>
        <inverted>false</inverted>
        <tagType>String;</tagType>
      </ruleNode>
    </conjNode>
    <conjNode type="And" errorMessage="">
    <ruleNode type="QuickFixRule" errorMessage="">
        <operation>Contains</operation>
        <tagNumber>49</tagNumber>
        <tagValue>1900</tagValue>
        <inverted>false</inverted>
        <tagType>String;</tagType>
      </ruleNode>
      <ruleNode type="QuickFixRule" errorMessage="">
        <operation>Equal</operation>
        <tagNumber>115</tagNumber>
        <tagValue>CLIENT1</tagValue>
        <inverted>false</inverted>
        <tagType>String;</tagType>
      </ruleNode>
    </conjNode>
  </conjNode>
</evaluator>
```

The code-based evaluator is represented by two C# code blocks, the first for client identification and the second for market identification. Whenever the user implements these code blocks, they are dynamically compiled into a temporary assembly, which is then used to retrieve the `ICodeEvaluator` interface instance. This interface declares two methods as shown in Example 8.7, and they are called to execute the corresponding dynamically compiled code and return the requested identifiers.

Example 8.7 ICodeEvaluator interface declaration

```
public interface ICodeEvaluator
{
  /// <summary>
  /// Returns the market identification code from the message represented by given tag dictionary.
  /// </summary>
  /// <param name="messageTags">Tags representing the message.</param>
  /// <returns>Market identification code.</returns>
  string GetMarketCode(Dictionary<int, string> messageTags);

  /// <summary>
  /// Returns the client identification code from the message represented by given tag dictionary.
  /// </summary>
  /// <param name="messageTags">Tags representing the message.</param>
  /// <returns>Client identification code.</returns>
  string GetClientCode(Dictionary<int, string> messageTags);
}
```

The code-based evaluator was implemented as a dynamic code because sometimes the client and market recognition process is not a simple task, which can be described with a set of predefined rules and a more sophisticated logic have to be applied. These dynamic code blocks give the user the freedom to apply the most complex logic without any limitation. However the rule definition is a task designed for a system administrator, to simplify a code implementation, the `Helper` class was implemented. This class is automatically linked with each dynamically compiled assembly, so the user can take advantage of its predefined methods and use them within the code. In Example 8.8 you can see a definition of such code blocks and a `Helper` class usage within them.

Example 8.8 Identification rule's dynamic code

```
public class CLIENT : ICodeEvaluator
{
    public string GetMarketCode(Dictionary<int, string> tags)
    {
        string market = Helper.GetTagValue(tags, 100);
        return Helper.GetMarketCode(market);
    }

    public string GetClientCode(Dictionary<int, string> tags)
    {
        string compID = Helper.GetTagValue(tags, 115);
        string subID = Helper.GetTagValue(tags, 116);
        return compID + "_" + subID;
    }
}
```

8.1.4 Communication

Monitoring server implements two communication layers, one for communication with systems, which comprise the electronic trading infrastructure and one for publishing the processed messages to the clients. The server communication architecture is outlined in Figure 8.4.

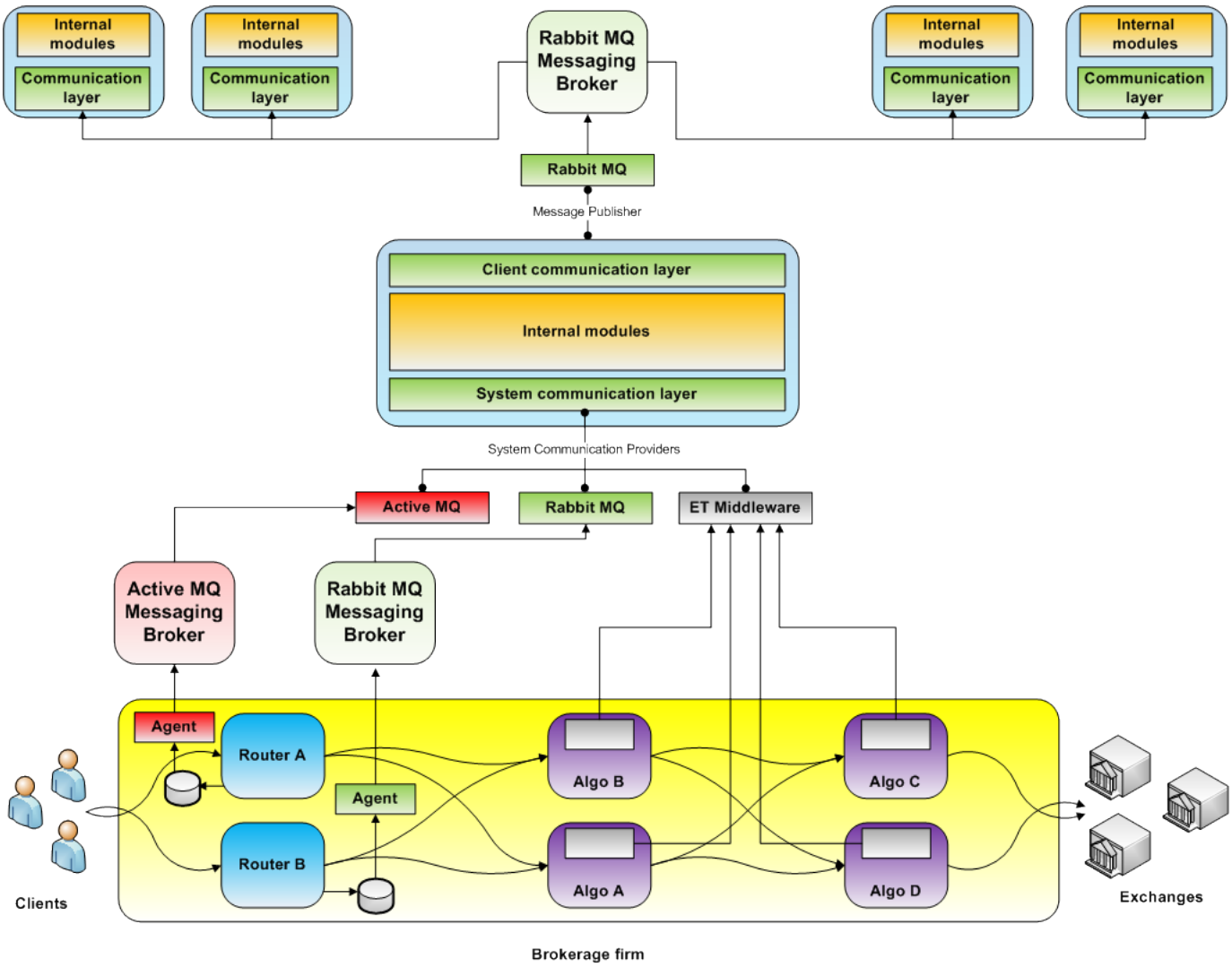


Figure 8.4: Monitor server communication architecture

As you can see, the more sophisticated layer is a system communication layer. Similar to application framework, it was important to implement a system communication layer in such way, that it will be independent of currently used middleware and will allow the communication with trading infrastructure systems through different types of middleware at the same time. This requirement was really important, because it is not very likely, that all systems throughout the whole trading infrastructure will use the same communication middleware in the future. As already discussed in Chapter 3, the monitoring system should support both the infrastructure with obsolete systems like FIX bridge and others and a future infrastructure formed especially by distributed algorithmic engines. In Figure 8.4 is illustrated a scenario, where these two infrastructures coexist and which is a current situation in sponsoring company. The router systems are the obsolete ones and communicate with other systems through FIX protocol, whilst algorithmic engine machines are using for their communication an internal electronic trading middleware and a customized protocol which is based on a FIX.

The other thing, which is important to note here is, that the obsolete and third-party systems can't be modified very easily and therefore there is no way to integrate the message publishing mechanism directly into them as it was done in case of algorithmic engines. This leads to an implementation of monitoring agents.

These agents are simple boxes, which monitor the system's persistence storage, where the whole system communication is stored, parse and transform its content into general structures used also by algorithmic engines and publish them to a corresponding monitoring server provider.

All currently used obsolete and third-party systems use for their communication a FIX protocol and persist all FIX messages to a log file, which is used as main persistence storage. Therefore there was only one monitoring agent implemented so far, which monitors the persistence file changes through `SystemFileWatcher` instance and whenever the file changes, it reads the newly added content, parses it with regular expression, which matches the FIX protocol messages, transforms these messages into corresponding structures present in `WAC.ET.Messages` namespace and publishes them to a RabbitMQ communication provider. The transformation process of FIX messages into general structures can be done either on agent side or on monitoring server side within the implementation of corresponding system communication provider. In current solution, it was decided to do this on agent side, but it does not matter where it is implemented for future components. The reason to do such transformation at all is to unify the message processing by the rest of server side components.

Now when you know why it was so important to design a system communication layer in as much flexible way as possible, we will describe how to integrate and implement required system communication providers. All important classes are implemented in `WAC.ET.Monitor.Server.Core.Communication` namespace and the main class responsible for communication providers' registration is a `SystemCommunicationManager`. It discovers and loads all available communication providers bound to extension point, which declaration is shown in Example 8.9.

Example 8.9 System communication providers extension point declaration

```
<ExtensionPoint path = "/Monitor/Communication/SystemProviders">
  <ExtensionNode name="SystemCommunicationProvider"
    type="WAC.ET.Monitor.Server.Core.Communication.SystemCommunicationProviderExtensionNode">
    <Description>Registers a new system communication provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

Corresponding extension node used to initialize the underlying provider instance declares same attributes as many other previously mentioned extension nodes, the `type` and `configFileName`. Each underlying provider has to implement an `ISystemCommunicationProvider` interface, which declaration is shown in Example 8.10.

Example 8.10 ISystemCommunicationProvider interface declaration

```
public interface ISystemCommunicationProvider : IComponent, IMessageSender
{
  /// <summary>Communication type identifier.</summary>
  string CommunicationType { get; }

  /// <summary>Sends a message to a system with given identifier.</summary>
  /// <param name="systemId">System identifier, to which the message should be sent.</param>
  /// <param name="message">Message to be sent.</param>
  void Send(string systemId, object message);

  /// <summary>
  /// Registers a new system, which should be monitored by this type of communication provider.
  /// </summary>
  /// <param name="systemId">System identifier, which should be monitored.</param>
  /// <param name="systemConfiguration">System specific communication configuration.</param>
  /// <param name="receiver">Message receiver, to which should be passed all received messages.</param>
  void Register(string systemId, string systemConfiguration, IMessageReceiver receiver);

  /// <summary>Creates system communication configuration instance from a given string.</summary>
  /// <param name="configurationString">Configuration string.</param>
  /// <returns>System communication configuration instance</returns>
  CommunicationProviderConfiguration CreateConfiguration(string configurationString);
}
```

As you can see, the `ISystemCommunicationProvider` interface is derived from `IMessageSender` interface and one of the parameters of its `Register` method is an `IMessageReceiver` instance. These two interfaces, `IMessageSender` and `IMessageReceiver`, are very important because they are used to pass messages between independent server components and allow to create a chain of message processors. The `IMessageSender` is used to tell the component to start sending messages and to register a next `IMessageReceiver` in processing chain. The `IMessageReceiver` is then used to process the received messages. Declarations of both interfaces are shown in Example 8.11 and Example 8.12.

Example 8.11 IMessageSender interface declaration

```
public interface IMessageSender
{
  /// <summary>Gets the state of the sender instance.</summary>
  SendingState SendingState { get; }

  /// <summary>Raised when the state of sending changes.</summary>
  event SendingStateChangedDelegate SendingStateChanged;

  /// <summary>Sets the receiver to which should be passed all received messages.</summary>
  IMessageReceiver MessageReceiver { set; }

  /// <summary>Tells the sender that it can start sending the messages.</summary>
  /// <remarks>The sending starts from a successor message after the one provided.</remarks>
  /// <param name="lastReceivedMessageIdentifier">The identifier of last received message.</param>
  void StartSending(GenericMessageIdentifier lastReceivedMessageIdentifier);

  /// <summary>Tells the sender to stop sending the messages.</summary>
  void StopSending();
}
```

Example 8.12 IMessageReceiver interface declaration

```
public interface IMessageReceiver
{
  /// <summary>
  /// Proccesses the given message.
  /// </summary>
  /// <param name="message">Message to be processed.</param>
  void ProcessMessage(GenericMessageEncapsulator message);
}
```

What was not noted so far is that each `ISystemCommunicationProvider` can be used to monitor several trading infrastructure systems, which use the same communication middleware for message publishing. To allow this functionality, the `ISystemCommunicationProvider` interface declares a `Register` method telling the provider instance, that it should monitor the system with given identifier and initialize its connection with given configuration. To better understand the previous paragraphs, the Figure 8.5 illustrates the communication provider's architecture and the role of `IMessageSender` and `IMessageReceiver` components in communication process.

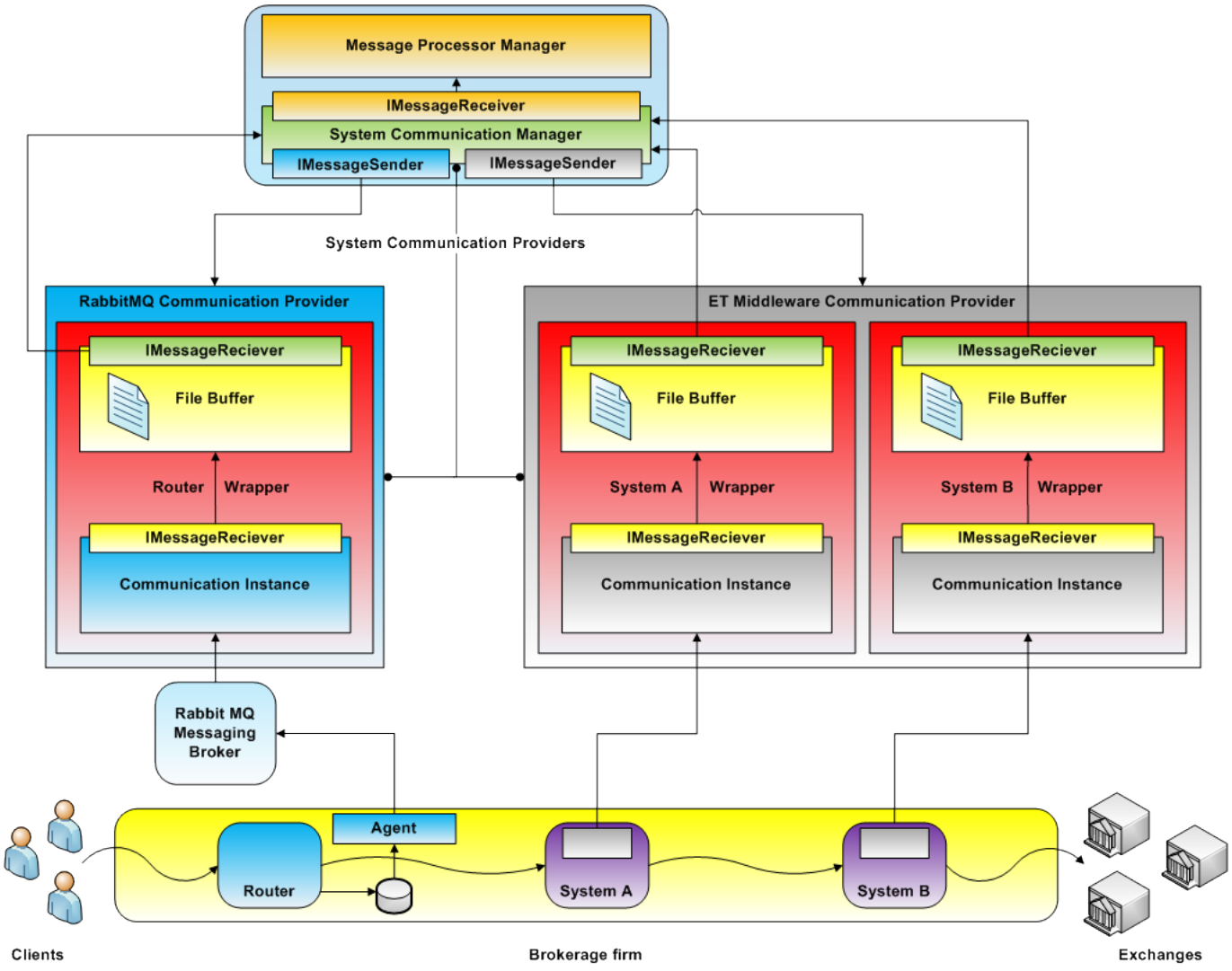


Figure 8.5: System communication provider architecture

As you can see, each communication instance related to exactly one trading system passes the received messages into a special component called `FileBuffer`. This component is used to persist each received message into a file, which is then used during the recovery process in case of monitoring system failure. The reason to store messages immediately when they are received is to ensure, that any already received message is not lost in case of system failure. Here comes the question, why are messages persisted in the file and not in a database, where they have to be stored anyway because of the history support? The reason to choose a file for immediate persistence was, that writing to a file is much faster than inserting into database and therefore allows processing messages in more real-time manner.

Besides the system communication layer, the monitoring server contains also a client communication layer, which is responsible for publishing the already processed messages to interested client terminals. This layer is much simpler and consists only of a message publisher provider, which is also implemented as a server external module. It is bound into an extension point, which declaration is shown in Example 8.13.

Example 8.13 Message publisher extension point declaration

```
<ExtensionPoint path = "/Monitor/Communication/MessagePublisher">
  <ExtensionNode name="MessagePublisher"
    type="WAC.ET.Monitor.Server.Core.Communication.MessagePublisherExtensionNode">
    <Description>
    Registers a new message publisher responsible for publishing all processed messages to clients.
    </Description>
  </ExtensionNode>
</ExtensionPoint>
```

The bound message publisher have to implement an `IMessagePublisher` interface, which as shown in Example 8.14 declares besides the common methods of `IComponent` interface only one `Publish` method. This method is used to tell the publisher, that the provided message should be published to the list of interested subscribers. The component responsible for `IMessagePublisher` instance registration is a `MessagePublisher`. This component is implemented as one of message processors and therefore will be described separately in the Section 8.1.5.6.

Example 8.14 IMessagePublisher interface declaration

```
public interface IMessagePublisher : IComponent
{
  /// <summary>Publishes the given message to a list of interested subscribers.</summary>
  /// <param name="interestedSubscribers">The list of interested subscribers.</param>
  /// <param name="message">The message, which should be published.</param>
  void Publish(HashSet<string> interestedSubscribers, MonitorMessage message);
}
```

8.1.5 Message processors

Now when you know, how the monitoring server receives the messages from trading infrastructure systems, we will describe in the following sections how these messages are processed. As already illustrated in Figure 8.5, the first component, to which are the messages passed directly from the `SystemCommunicationManager`, is a `MessageProcessorManager`. This component is implemented as an extensible module, which allows binding of different types of `MessageProcessor` instances and building message processing chain. The registration of new `MessageProcessor` instance is implemented through extension point, which declaration is shown in Example 8.15.

Example 8.15 Message processor extension point declaration

```
<ExtensionPoint path = "/Monitor/MessageProcessors">
  <ExtensionNode name="MessageProcessor"
    type="WAC.ET.Monitor.Server.Core.MessageProcessing.MessageProcessorExtensionNode">
    <Description>Registers a new message processor component.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

Each registered `MessageProcessor` has to implement an `IMessageProcessor` interface shown in Example 8.16. It derives from an `IComponent` interface, so it declares standard methods for component's initialization, start-up and uninitialization and also derives from `IMessageReceiver` interface described in previous section, so it can process the incoming messages. Each `MessageProcessor` contains also a reference to a next processor in message processing chain, so the processed message can be passed to it.

Example 8.16 IMessageProcessor interface declaration

```
public interface IMessageProcessor : IComponent, IMessageReceiver
{
  /// <summary>Instance of next message processor in processing chain.</summary>
  IMessageProcessor NextMessageProcessor { get; set; }

  /// <summary>
  /// Tells the processor to remove the previously processed message.</summary>
  /// <param name="message">Message to be removed.</param>
  void RemoveMessage(GenericMessageEncapsulator message);
}
```

Once the `MessageProcessorManager` loads and initialize all currently available `MessageProcessor` instances, it builds the message processing chain from them. The build process is fully configurable through configuration file passed to `MessageProcessorManager` during its initialization, so the system administrator can choose which message processors will be included in the message processing chain and in which order they will process the incoming messages. This is quite important, because each message processor can enrich the processed message by a set of additional properties, which are then required by other processors in the chain. During the implementation it was decided, that the most suitable representation of message processor chain will be a directed acyclic graph, where each node represents a `MessageProcessor` instance and each edge between two nodes represents a dependency between two processors. The reason to choose a directed acyclic graph for chain representation was that the graph can be topologically sorted and therefore the dependencies between message processors can be easily defined. The other advantage of such representation is, that it allows to create a set of independent message chains through a not connected graph and also to find processors, which are not influenced and can continue with message processing in case there occurs any exception. To better understand the previous text, let's see the Figure 8.6.

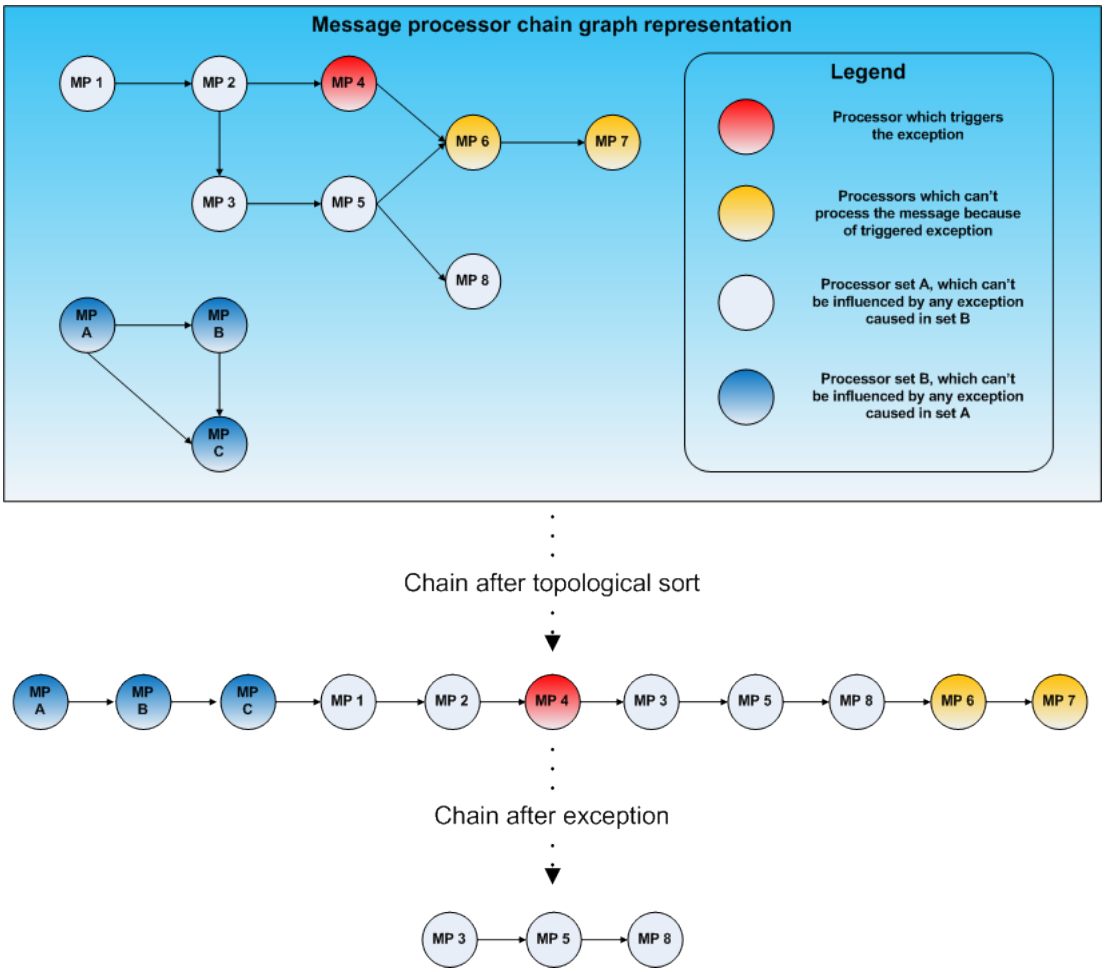


Figure 8.6: Message processor chain example

In illustrated example you can see, that the message processor chain is represented by a not connected graph, which defines two independent groups of message processors, that can't influence each other in case of process exception. With such decomposition, it is possible to define a set of message processors, which are logically independent and where each group is responsible for different task. For example, one group can be used for monitoring purposes and the second one can be used for publishing messages to other systems, which need the information about message flow. In the graph you can also see, that during the message processing, the message processor 4 fails, but this failure influences only message processors 6 and 7, so the message can be still passed to message processor 3 and from there to the following processors 5 and 8.

As mentioned in a previous paragraph, each message can be enriched during its processing by a set of dynamic properties. To allow such functionality, each message received from a trading system is immediately wrapped in a communication provider into a `GenericMessageEncapsulator` class, which contains the message itself and a couple of additional properties, from which the most important ones are the following:

- **MessageIdentifier** - Unique identification of a message within the whole monitoring system. It contains information from what trading system the message was received, on which trading day and what message number was assigned to it.

- **DynamicProperties** - A dictionary of all dynamic properties added by message processors during the message processing. Each additional property has to be added as a pair of unique string, which identifies the property and the property itself. The following message processors then can retrieve these properties and use them to correctly process the received message. All these additional properties are then published also to monitoring clients, so they can use them for message visualization.
- **DatabaseObjects** - A dictionary of all dynamic database objects added by message processors during the message processing. Each database object has to be added as a pair of unique string, which identifies the object and the object itself. The following message processors then can retrieve these objects and use them to correctly process the received message. All these database objects are finally stored by a `DatabaseSaver` component into a database and therefore they should completely represent a message, so it can be retrieved later by historical queries.
- **IsRecovery** - Flag indicating, that the message should be processed by message processors as a recovery message and therefore may be processed differently than during the real-time processing.
- **IsFake** - Flag indicating, that the message is a fake message. This kind of message is sent if you want to pause the message processing and ensure, that all message processors already processed all in-flight messages.
- **ReceiveTime** - Time, when the message was received.

Current implementation of monitoring server includes a set of message processors, which built together a message processing chain as illustrated in Figure 8.7 and their role and functionality will be described separately in the following sections.

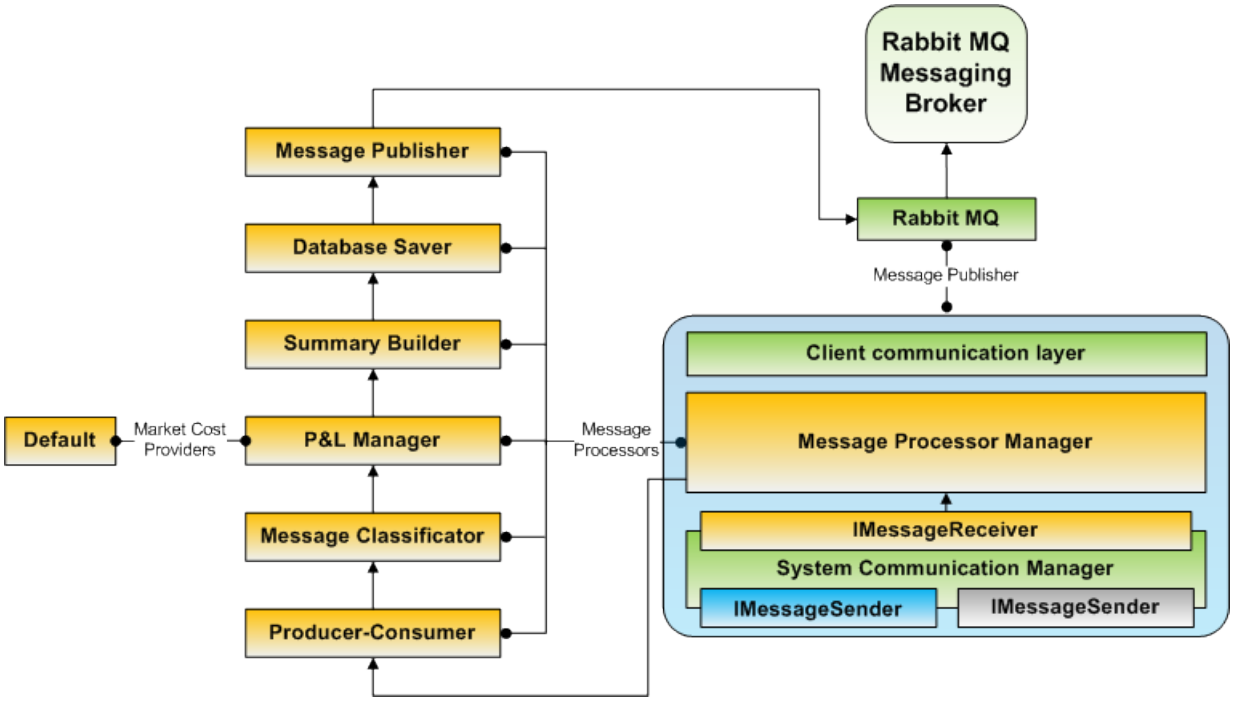


Figure 8.7: Chain of built-in message processors

8.1.5.1 Producer-consumer

The very first message processor in a message processor chain is a so called producer-consumer component, which implements a well known producer-consumer design pattern. Even though the component is really simple from the point of implementation view and that it doesn't process messages at all, it is very important and useful. The reason to implement it was to separate the producers (system communication providers in our case) from message consumers (following processors in a message processor chain). This separation was important because the messages can be produced by trading infrastructure in a much faster way then their actual processing by any message processor. If this component is not present in processor chain, the communication providers have to wait until each message is processed by all subsequent message processors and therefore have to block their message receiving process. The producer-consumer design pattern is suited exactly to solve such situations and it lets the consumer to process the data at its own pace, while allowing the producer to queue additional data at the same time. Figure 8.8 illustrates this component in action.

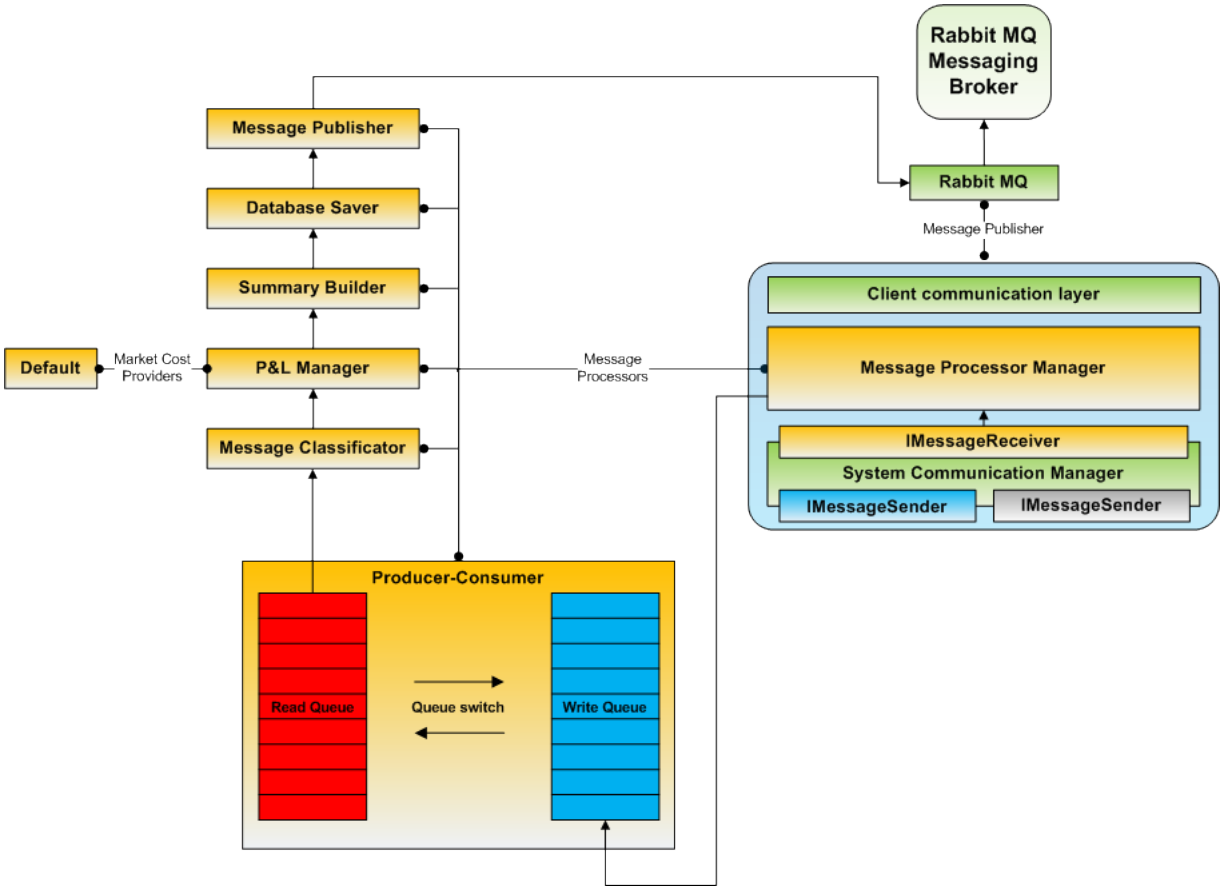


Figure 8.8: Producer-consumer integration

8.1.5.2 Message classificator

`MessageClassificator` can be supposed as a most important message processor. Before we describe its main functionality, the trading infrastructure and message flow process have to be explained in more detail, so the reader better understand the following text. The first thing, which has to be noted here is that

each system contains a set of sessions, which represent a communication channel between two systems of trading infrastructure or between the clients and trading infrastructure entry point. Each session is then from the system point of view either an input session or output session. Input sessions are sessions, defined on a so called client side of system and output sessions are defined on a market side of the system. The other type of session division is, that each session can be defined as a FIX session, which communicates with other systems through a FIX protocol or an internal session, which uses for its communication an internal electronic trading middleware. The Figure 8.9 illustrates these session types and how they can be used within the trading infrastructure.

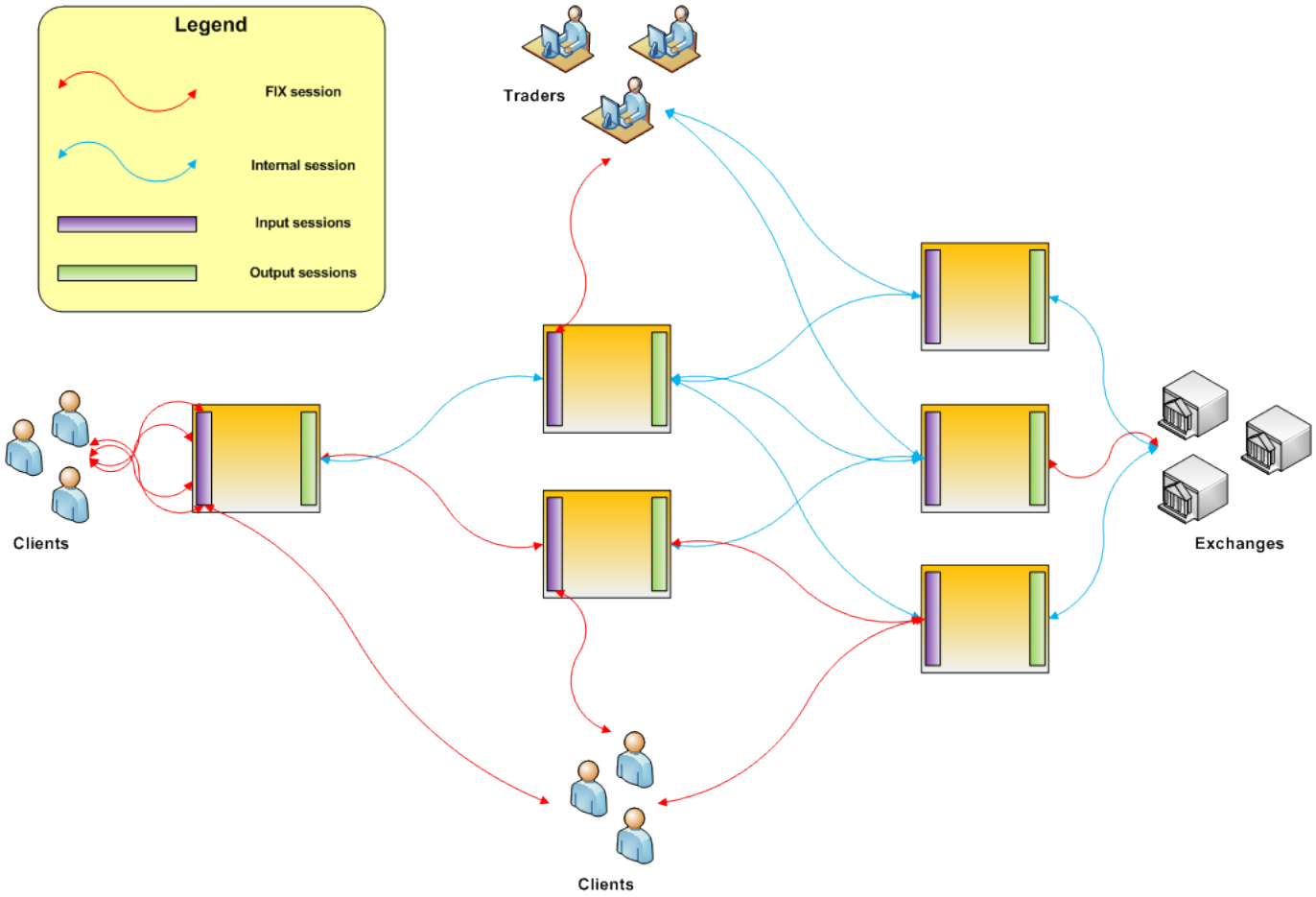


Figure 8.9: Trading system session types

Now when you know, how are the systems connected and what types of sessions can be used, we will describe, how the message flows throughout the trading infrastructure and what is a so called *message chain* and *message hierarchy*. Each incoming or outgoing message to or from an entry point of whole trading infrastructure is called a *client message*. The client message then flows through a set of trading systems based on routing rules defined on each of them. Each system involved in a message routing can modify message properties and even split the message into more messages. Such behaviour is very common in systems containing some algorithmic logic. It should be noted here, that whenever the system receives a message it creates its copy, modifies its properties if needed and sends a new message out of the system. In other words, each received *client message* results in a set of messages, which describe the *client message*'s path throughout the whole trading infrastructure. This set of messages is called *message chain* and each

message included in this chain except the *client message* is called *internal message*. Example of such *message chain* is shown in Figure 8.10.

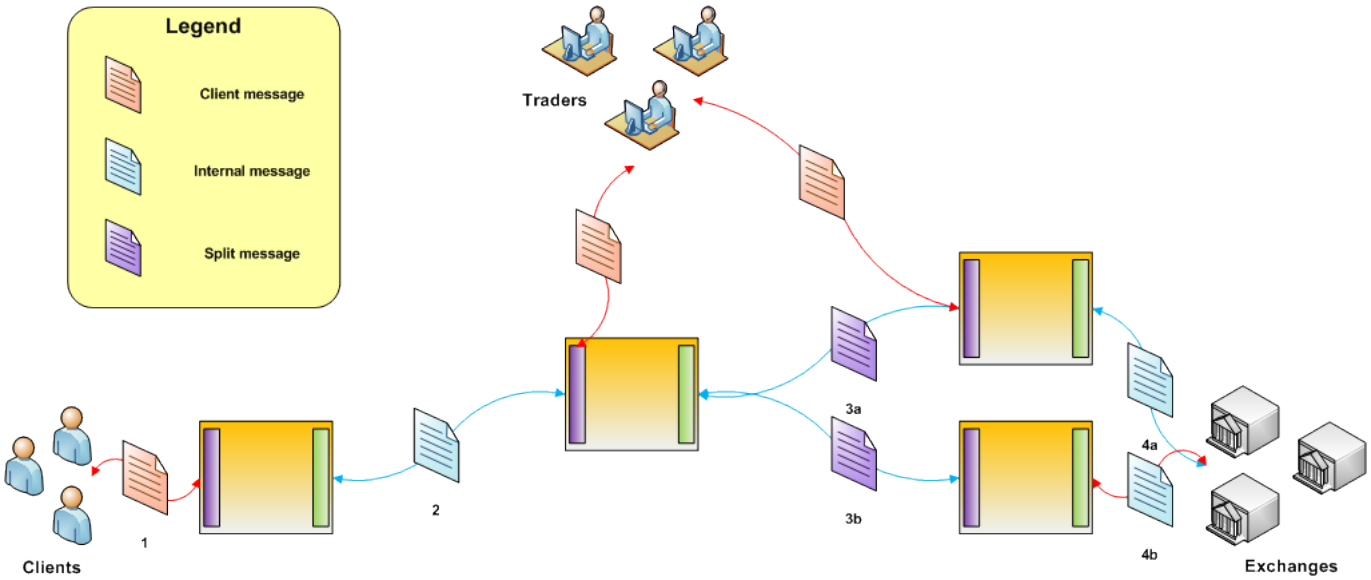


Figure 8.10: Message chain example

In Section 2.5 a couple of message scenarios were described. These scenarios apply to a one session only and it does not matter if it is a FIX or internal session. What was not noted there is, that all messages can be logically divided into two groups - *client messages* (do not confuse with *client message* from the message chain point of view) and *market messages*. *Client messages* are messages, which originate on a client side and therefore cover messages, which represent client requests, whilst *market messages* are messages originated on market side and represent the responses to the client requests. To properly connect each *market message* to a corresponding *client message*, the FIX protocol and therefore also a derived internal protocol defines a set of message tags, which are used for this purpose. Besides that each *market message* has to be connected to a *client message*, there are situations, when a client message has to be connected to a corresponding parent client message as well. An example of such situation is a replace or cancel request of an existing order. The result of this message referencing is that each order and its lifecycle can be described with a set of messages, which logically belong to this order and together build a *message hierarchy*. Example of such *message hierarchy* is illustrated in Figure 8.11.

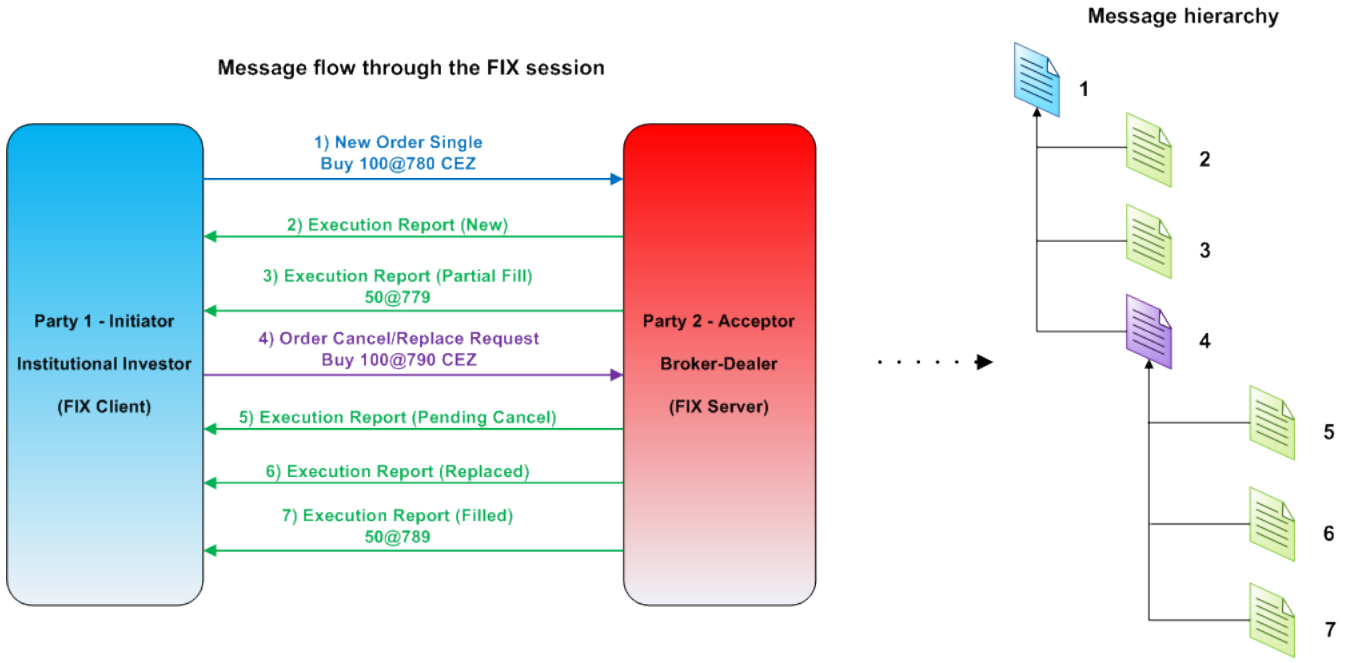


Figure 8.11: Message hierarchy example

Now when you know the most important terms of message processing, we can describe, what the main responsibilities of `MessageClassificator` component are:

- **Identification of fundamental message properties** - It is responsible for identification of which client sends the message and to what market is the message designated. To identify these two parameters, the `MessageClassificator` passes the received message to a `MessageIdentificationRules-Storage` component, which implements a message identification process described in Section 8.1.3. Besides the client and market identification, the component gather the information about from which system the message was published, through which FIX or internal session the message was received or sent, what is the message instrument and few other important properties.
- **Message errors detection** - It validates all messages against set of predefined rules and whenever an error is detected, the message is marked with a corresponding error type. The predefined rules check whether the client and instrument exist in the external storage, whether there exists an identification rule, which matches the message, whether the message is duplicate or not, whether the parent message in *message hierarchy* exists, whether the instrument identification can be retrieved from the message etc.
- **Message latency calculation** - One of the requirements for a monitoring system was to generate latency reports for each client. To allow creation of such reports, the latency of each *client message* has to be measured and this is done right in the `MessageClassificator` component.
- **Message hierarchy construction** - As discussed above, each order and its lifecycle can be described with a *message hierarchy*. The `MessageClassificator` is responsible to find references between particular messages and build such *message hierarchies*, so they can be used afterwards by other server and also client components.

- **Message chain construction** - `MessageClassificator` analyzes all messages from the whole trading infrastructure and constructs from them a corresponding *message chains*, so they can be used afterwards by other server and also client components.

8.1.5.3 P&L manager

P&L manager is a component responsible for calculation of revenues, costs and expenses of running the electronic trading business. Correct calculation of trading business P&L is quite complex process and therefore only basic principals will be described in this section. Basically each client has predefined a set of commissions that are applied for each executed trade on a given stock exchange. These commissions vary for for each stock exchange, for different types of services (e.g. CARE and DMA service) and represent revenues of trading business. On the other hand, each exchange and settlement centre defines a set of fees, which each market participant have to pay for trading and settlement process and these fees then represent direct costs of trading business. Whilst client commissions are defined within the scope of external client storage management and their definitions can be described with a set of common rules, the exchange and settlement fees definitions are very complex, vary with each market and can't be easily described in a general way. Therefore the P&L manager was implemented as an extensible component, to which the programmer can register a set of cost providers, where each of them implements a fee calculation process that corresponds to a given market. The next paragraph describes how the programmer can implement such cost provider and how it is registered with the monitoring server.

The core class of P&L manager component is a `PaLManager`, which implements an `IMessageProcessor` interface and is also responsible for discovering and initializing available cost providers. To allow the P&L manager component to be extended, it exposes an extension point, which declaration is shown in Example 8.17.

Example 8.17 Market cost providers extension point declaration

```
<ExtensionPoint path = "/PaLManager/MarketCostProviders">
  <ExtensionNode name="MarketCostProvider"
    type="WAC.ET.Monitor.Server.MessageProcessors.PaLManager.MarketCostProviderExtensionNode">
    <Description>Registers a new market cost provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

This extension point is used to register a set of cost provider instances, which are responsible for calculation process of market and settlement fees. Each cost provider has to derive from `IMarketCostProvider` interface. It derives from an `IMarketCostConfigurationProvider` interface, which defines methods to create and update the provider's specific configuration, through which the user can define fees, that should be used within the calculation process. The only provider specific method of `IMarketCostProvider` interface is a `CreateInstance` method used by `PaLManager` instance to tell the provider to create a new instance of `IMarketCostProviderInstance` interface. This created instance is the one, responsible for calculation process and it allows using one `IMarketCostProvider` instance for several markets. This is useful in situations when more markets define the same fee calculation rules and they only differ in the size of the fees. User then can use the same provider for these markets and for each of them define a different fees through the market specific provider configuration. Once the `PaLManager` is initialized and all available providers instantiated, then for each incoming message a corresponding provider is found and the message is passed to it through its `Calculate` method. This method calculates

the set of predefined cost types, so they can be included in a message in form of dynamic properties and used afterwards on the client side to display P&L summary.

Declaration of all important and above mentioned interfaces are shown in the following examples.

Example 8.18 IMarketCostConfigurationProvider interface declaration

```
public interface IMarketCostConfigurationProvider
{
  /// <summary>Gets name of the provider.</summary>
  string Name { get; }

  /// <summary>Creates configuration instance based on a given string.</summary>
  /// <param name="configurationString">Configuration string.</param>
  /// <returns>Configuration instance.</returns>
  MarketCostConfiguration CreateConfiguration(string configurationString);

  /// <summary>Updates the configuration.</summary>
  /// <param name="configuration">Configuration to apply.</param>
  void UpdateConfiguration(MarketCostConfiguration configuration);
}
```

Example 8.19 IMarketCostProvider interface declaration

```
public interface IMarketCostProvider : IMarketCostConfigurationProvider
{
  /// <summary>Indicates whether the provider is initialized or not.</summary>
  bool IsInitialized { get; }

  /// <summary>
  /// Creates a cost provider instance for a given market and configures it based on a given ↩
      configuration.
  /// </summary>
  /// <param name="marketMICCode">Market MIC code.</param>
  /// <param name="configuration">Provider's configuration.</param>
  /// <returns>Cost provider instance.</returns>
  IMarketCostProviderInstance CreateInstance(string marketMICCode, MarketCostConfiguration ↩
      configuration);

  /// <summary>Initializes the provider.</summary>
  /// <param name="configuration"></param>
  void Initialize(string configuration);

  /// <summary>Uninitialize the provider.</summary>
  void Uninitialize();
}
```

Example 8.20 IMarketCostProviderInstance interface declaration

```
public interface IMarketCostProviderInstance
{
  /// <summary>Default trading currency of the provider.</summary>
  string TradingCurrency { get; set; }

  /// <summary>Calculates the market costs related to the given message.</summary>
  /// <param name="message">Message which participates in P&L calcaluation.</param>
  /// <param name="executionInfo">Common information about the execution.</param>
  /// <param name="orderInfo">Common information about the corresponding order.</param>
  /// <returns>Market costs related to the given message.</returns>
  PaLUpdateInfo Calculate(GenericMessageEncapsulator message, PaLExecutionInfo executionInfo,
      PalOrderInfo orderInfo);

  /// <summary>Initializes the provider.</summary>
  void Initialize();
}
```

8.1.5.4 Summary builder

Summary builder is used to create customizable views, which summarize all or a subset of currently processed messages and their properties, and provides these views to client terminals or to other server components. The reason to implement this component was to give the user an overview about what happens within the trading infrastructure. Because there are plenty of different views, which can be useful for different types of analysis, it was important to implement the summarization process as flexible as possible, so the users can customize these views for their specific needs. The following paragraphs will describe how can be these views defined and how they are constructed.

There are three important objects, which are used to describe the summary view instance. It is a summary filter, summary group and summary view itself. The relationship between these three objects is illustrated in Figure 8.12.

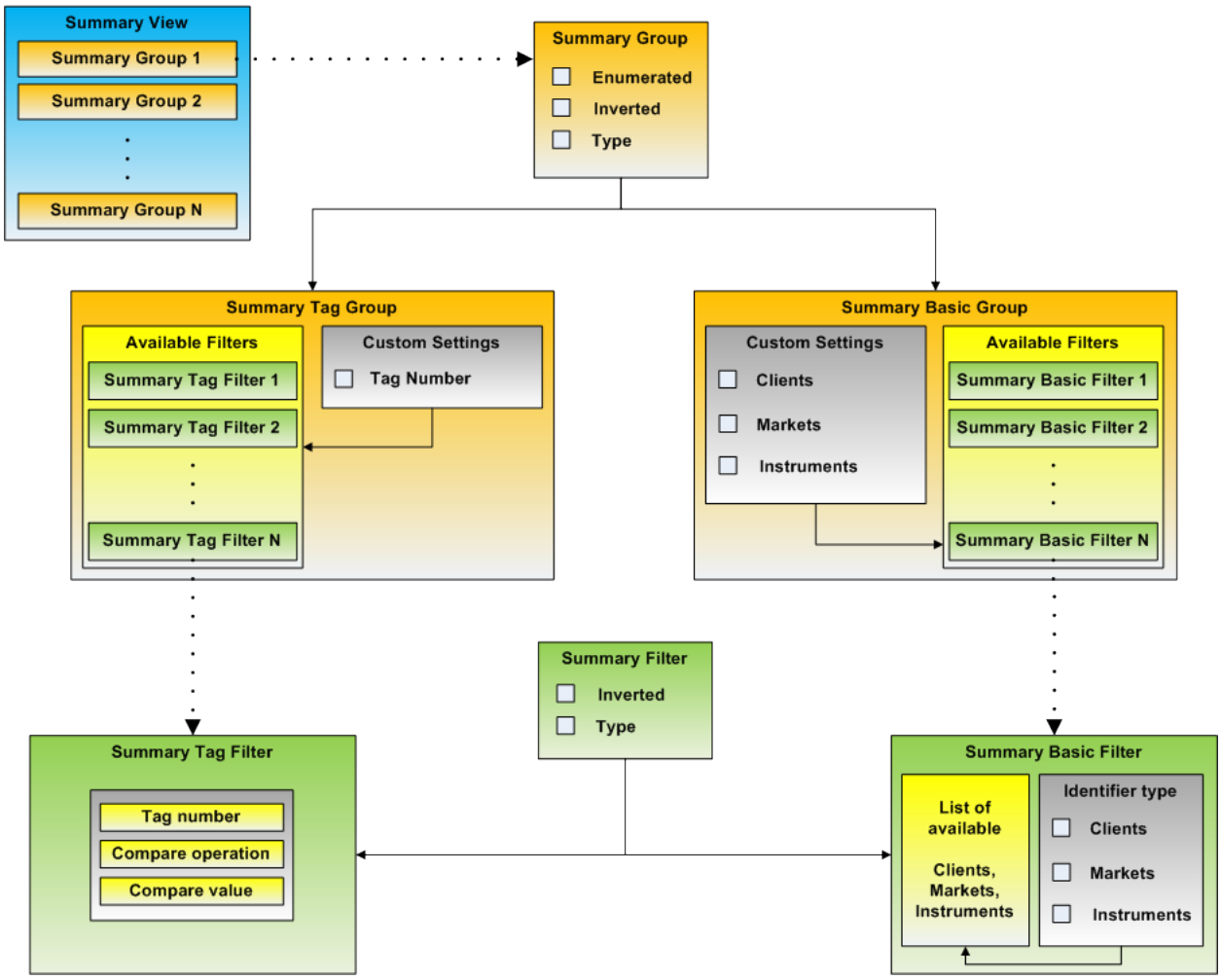


Figure 8.12: Summary view definition objects relationship

As you can see, each `SummaryView` instance consists of a set of `SummaryGroup` instances, where the position of the group within the summary view determines the level in group hierarchy. There are two types of `SummaryGroups` implemented, the `SummaryTagGroup` represented by a message tag based values and the `SummaryBasicGroup` represented by a basic properties of given messages. The list of values of both types of `SummaryGroup` object can be represented in two ways depending on whether the group is *enumerated* or not. The *enumerated* group is represented by all unique values, which occur in a specified tag or a specified basic property, whilst *not enumerated* group consists of names of included `SummaryFilter` instances. The `SummaryFilter` object is then used to group the set of messages with the same value either in the given tag or in a given basic property depending on what type of filter is used.

The most important class, which is responsible for building the summary view is a `SummaryViewBuilder`. It builds for each received message a so called *summary path* and calculates the data, for each summary column. The *summary path* is used to represent unique path in a summary hierarchy and corresponds to a summary view row. It is then used by the component, which visualizes the summary view to determine, to which row of the summary view the calculated data should be applied. The process of building the *summary path* is as follows: with each message the builder loops through all summary groups included in a given view and for each group it resolves its value. These values are then concatenated together and this concatenation represents the summary view hierarchy path. The Figure 8.13 illustrates the summary view,

which is represented by three `SummaryBasicGroup` instances, which are all enumerated. The first group is enumerated over a *Client* property, the second over a *Market* property and the last one over an *Instrument* property.

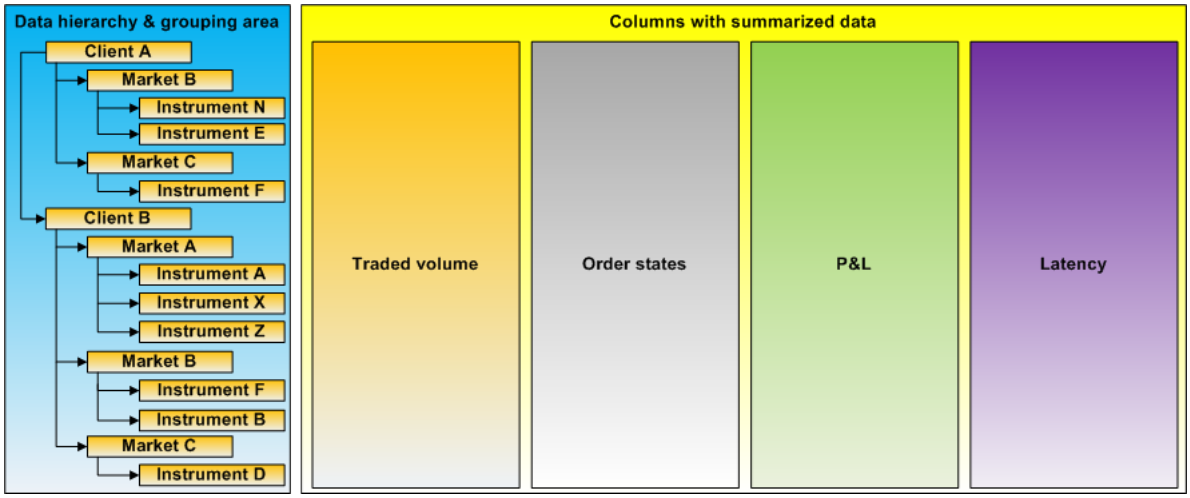


Figure 8.13: Summary view hierarchy and path example

As you can see, the current implementation of summary view contains many columns, which are calculated during the build process and can be customized by the user. All columns are divided into four logical groups that are as follows:

- **Traded volume** - Columns containing information about how many shares were bought or sold on a given summary path and what were the average prices and total traded volume.
- **Order states** - A large group of columns, which tell the user how many orders are in a specific trading state at a given time. This gives the user immediate overview about how many open orders are on the market, how many orders were filled, rejected, suspended etc.
- **P&L** - This group of columns contains summarized information about revenues and costs, so the user can see, what are the client commissions, what have to be paid to a settlement centre, what are the fees to stock exchanges etc.
- **Latency** - Columns, which summarize the average latencies for a specific message types.

To get better picture about summary view representation and about the role of corresponding objects, please see the [Postulka10] included on the attached CD, where is the whole process of summary view definition explained from the user point of view.

8.1.5.5 Database saver

Database saver is a very simple message processor, which is responsible for database persistence of each processed message. Whenever it receives a message, it takes all database objects that were added by previous message processors, and pass them to an `AsynchronousDatabaseSaver` instance. This `AsynchronousDatabaseSaver` then stores received database objects in an asynchronous way in specified

time intervals. The reason to use the asynchronous saving process is, that such implementation can't slow down or even block the real-time message processing, which can easily happen if the synchronous implementation is used. Databases are in general not designed for real-time applications and therefore their performance is not sufficient. Because the monitoring server do not need the database message persistence for real-time message processing and the persistence is used for historical queries only, the asynchronous implementation is the most suitable one. Monitoring server administrator can configure a couple of database saver component's properties, such as a size of transaction which is used during the saving process, the time interval between each attempt to save newly added database objects and the number of queued objects, which triggers the component to start warning about insufficient database performance. Component's role in message processor chain is illustrated in Figure 8.14.

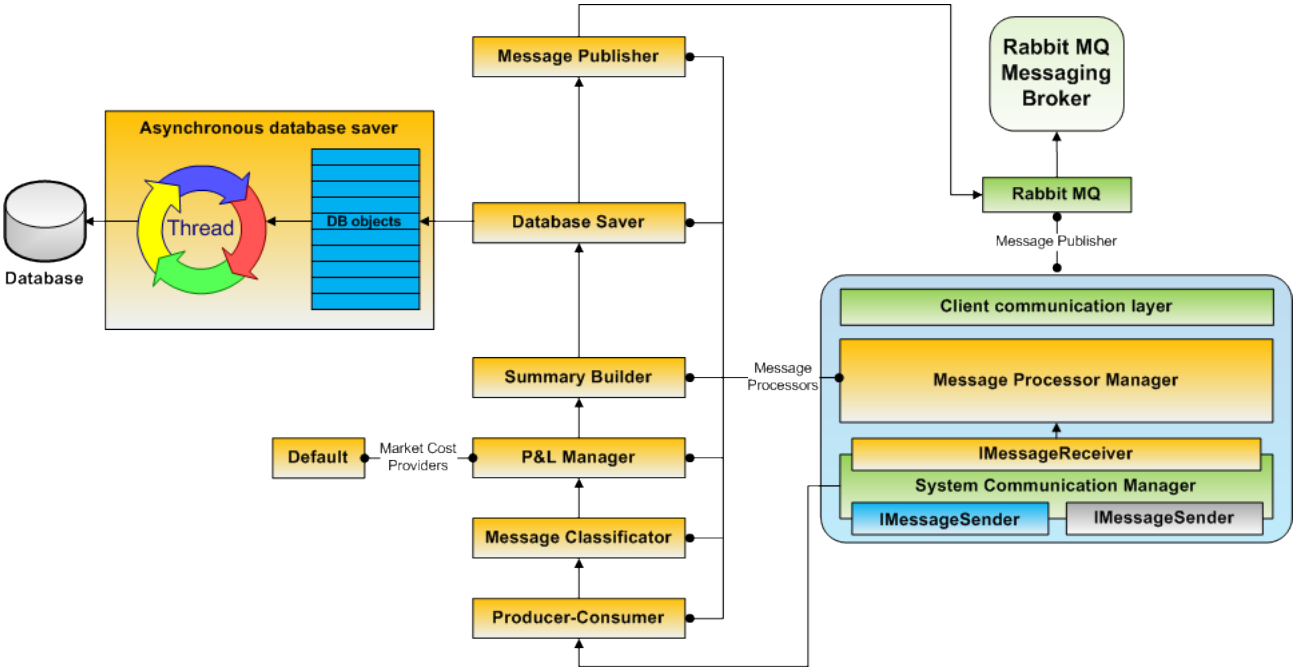


Figure 8.14: Asynchronous database saver's role in message processing

8.1.5.6 Message publisher

Message publisher is a last component in a message processor chain and is responsible for publishing the processed messages to interested clients. To better understand how the message publisher figures out, which clients are interested in a particular message, the following paragraph describes the process of message subscription and the role of message filters in it.

Whenever the user wants to display a set of messages on client side, the `MessageSubscriptionRequest` is sent to the server. Each request contains a subscriber's identifier, which should be used during the message publishing, the subscription identifier, which is used to match the subscription response on the client side, the `MessageFilter` describing in what messages the subscription is interested and optionally a `SnapshotInfo` instance, which can be used to specify custom properties for subscription snapshot data. The server component listening for `MessageSubscriptionRequests` is a `MessagePublisher`. It is a core class in message publishing process and its responsibility is to manage all subscriptions and evaluate the received messages against them.

The most important role in message publishing process play message filters, which describe the set of messages, in which is the client interested. Each message filter can be described by the following three parts:

- **Message classification filter** - Filter used to match basic classification properties of `MonitorMessage` such as client, market, instrument, system and session identifiers and few others.
- **Message tag filter** - A tag-based filter used to match a wrapped raw message included in each `MonitorMessage` instance. Filter is represented in a same way as a tag-based evaluator mentioned in Section 8.1.3.
- **List of dynamic filters** - As mentioned in the beginning of the Section 8.1.5, each message processor can add to a currently processed message a set of custom objects. To allow the user to filter messages based on these dynamic properties, a dynamic filter can be implemented and added to the list of dynamic filters, which are then used within the filtering process.

Each dynamic filter has to implement an `IMessageFilter` interface same as it do also `MessageClassificationFilter` and `MessageTagFilter` instances. Through this interface the programmer specifies, on which type of object included in a `MonitorMessage` instance should be this filter applied and implements matching and equality methods. The declaration of `IMessageFilter` interface is shown in Example 8.21.

Example 8.21 IMessageFilter interface declaration

```
public interface IMessageFilter
{
  /// <summary>Object type on which this filter can be applied.</summary>
  string SupportedObjectType { get; }

  /// <summary>Indicates whether the filter is empty or not.</summary>
  /// <remarks>Checks whether the message should be matched against this filter or not.</remarks>
  bool IsEmpty { get; }

  /// <summary>Indicates whether the given filter is equal to this instance.</summary>
  /// <param name="filter">Filter which should be checked for equality.</param>
  /// <returns><c>true</c> if filters are equal, otherwise <c>false</c></returns>
  bool IsEqual(IMessageFilter filter);

  /// <summary>Checks whether the filter matches the given object.</summary>
  /// <param name="objectToMatch">Object to be matched by filter.</param>
  /// <returns><c>true</c> if the filter matches the object, otherwise <c>false</c></returns>
  bool Matches(object objectToMatch);

  /// <summary>Initializes the filter.</summary>
  void Initialize();
}
```

Because several clients can be interested in same set of messages at the same time it wouldn't be efficient to match each message to all received message filters. The better solution was to save only one instance of each unique message filter and a corresponding list of interested subscribers, so the message has to be matched only once. To decide whether the message filter instance is unique or not, the `Equals` method of `IMessageFilter` interface is used.

Besides the main purpose of subscription request, to real-time filter and deliver messages to the interested clients, each response for a subscription request has to contain snapshot data, which are used on the client

side to visualize the current monitoring state. Because the number of already processed messages at time of subscription receipt can be very high, it was important to optimize the process of snapshot creation. For this purpose the `FilterStorageManager` class was implemented. This class allows each `IMessageFilter` instance to register a custom and optimized message storage, which should be used to retrieve all snapshot messages matching the given filter. Each registered storage has to implement an `IFilterMessageStorage` interface and its declaration is shown in Example 8.22.

Example 8.22 IFilterMessageStorage interface declaration

```
public interface IFilterMessageStorage
{
  /// <summary>
  ///  Filters the given list of messages based on a given filter its application level.
  /// </summary>
  /// <param name="applicationLevel">
  ///  Indicates whether the filter should be applied on all messages or only on orders.
  /// </param>
  /// <param name="filter">Filter used to filter the messages.</param>
  /// <param name="messages">List of messages, which should be filtered.</param>
  /// <returns>List of already filtered messages.</returns>
  List<MonitorMessage> GetMessages(MessageFilterApplicationLevel applicationLevel, IMessageFilter ↩
      filter, List<MonitorMessage> messages);

  /// <summary>Initializes the storage.</summary>
  void Initialize();

  /// <summary>Stores the given message into the storage.</summary>
  /// <param name="message">Message to be stored.</param>
  void StoreMessage(MonitorMessage message);

  /// <summary>Removes the given message from the storage.</summary>
  /// <param name="message">Message to be removed.</param>
  void RemoveMessage(MonitorMessage message);
}
```

Because the snapshot data can be represented in many forms depending on a component, which creates the message subscription, it was also important to support the registration of custom snapshot providers. These providers are registered directly with the `MessagePublisher` instance and used whenever the subscription request contains an optional `SnapshotInfo` property. The whole process of snapshot retrieving is shown in Figure 8.15.

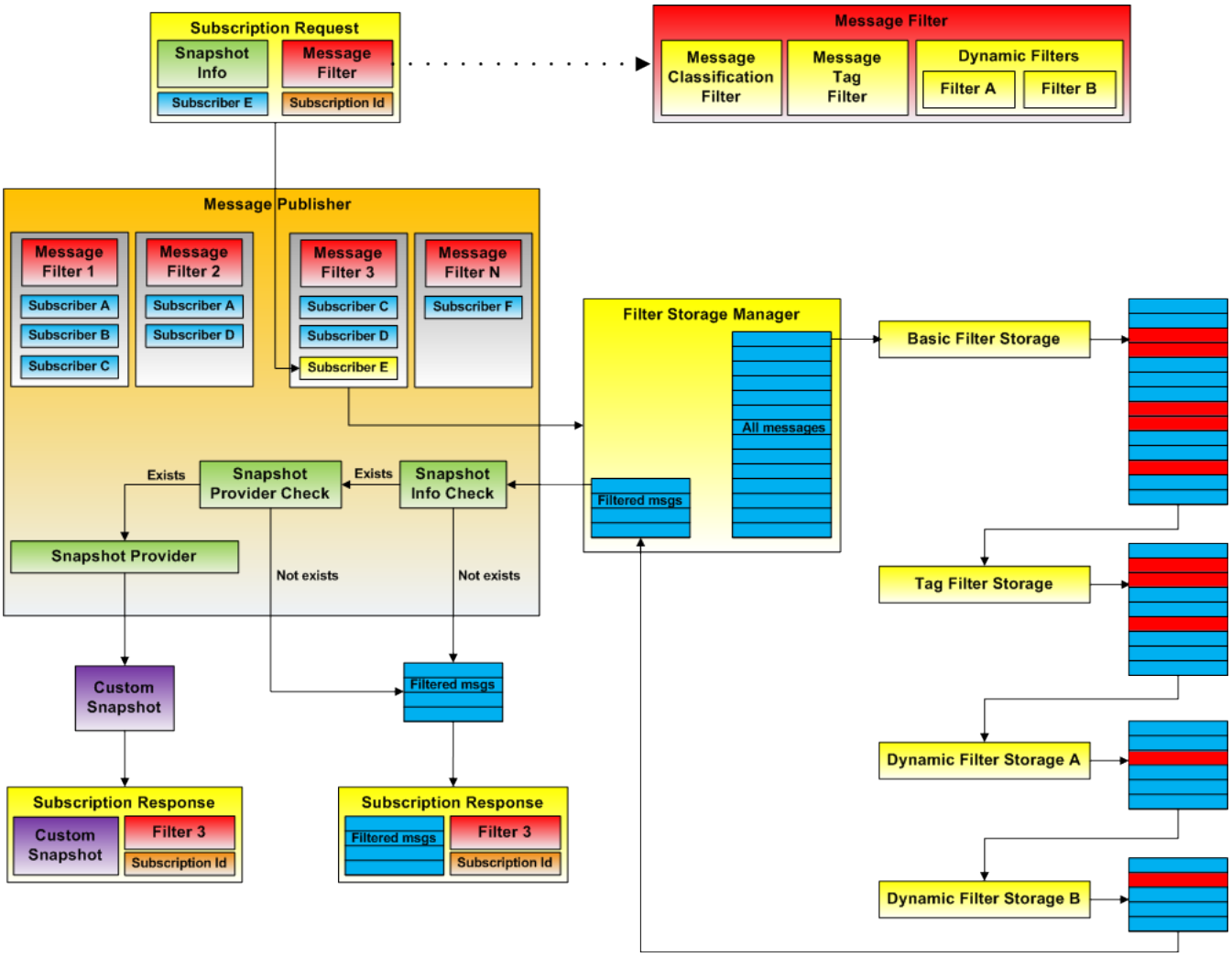


Figure 8.15: Message subscription snapshot retrieving process

The `MessagePublisher` passes the received `MessageFilter` to a `FilterStorageManager`, which takes all up to that time processed messages and subsequently passes them to all custom storages, that correspond to the list of `IMessageFilter` instances included in a given `MessageFilter` instance. Each custom storage removes the messages, which do not match a related filter from the passed message collection and this modified collection is passed to the next custom storage. Once the message snapshot filtering process is ended, the resulted messages are returned to the `MessagePublisher`, which checks whether the subscription request contains a `SnapshotInfo` property. In case the `SnapshotInfo` property is present, the `MessagePublisher` tries to find a corresponding `IMessageSubscriptionSnapshotProvider` and in case it exists, it passes the filtered snapshot messages to this provider. The provider then can transform these messages into a custom object and this object is returned as a snapshot for received subscription. In case the `SnapshotInfo` property is not included in the subscription request or in case the corresponding snapshot provider doesn't exist, the snapshot is represented by a list of filtered messages.

The real-time message publishing process, which follows the message subscription, is illustrated in Figure 8.16.

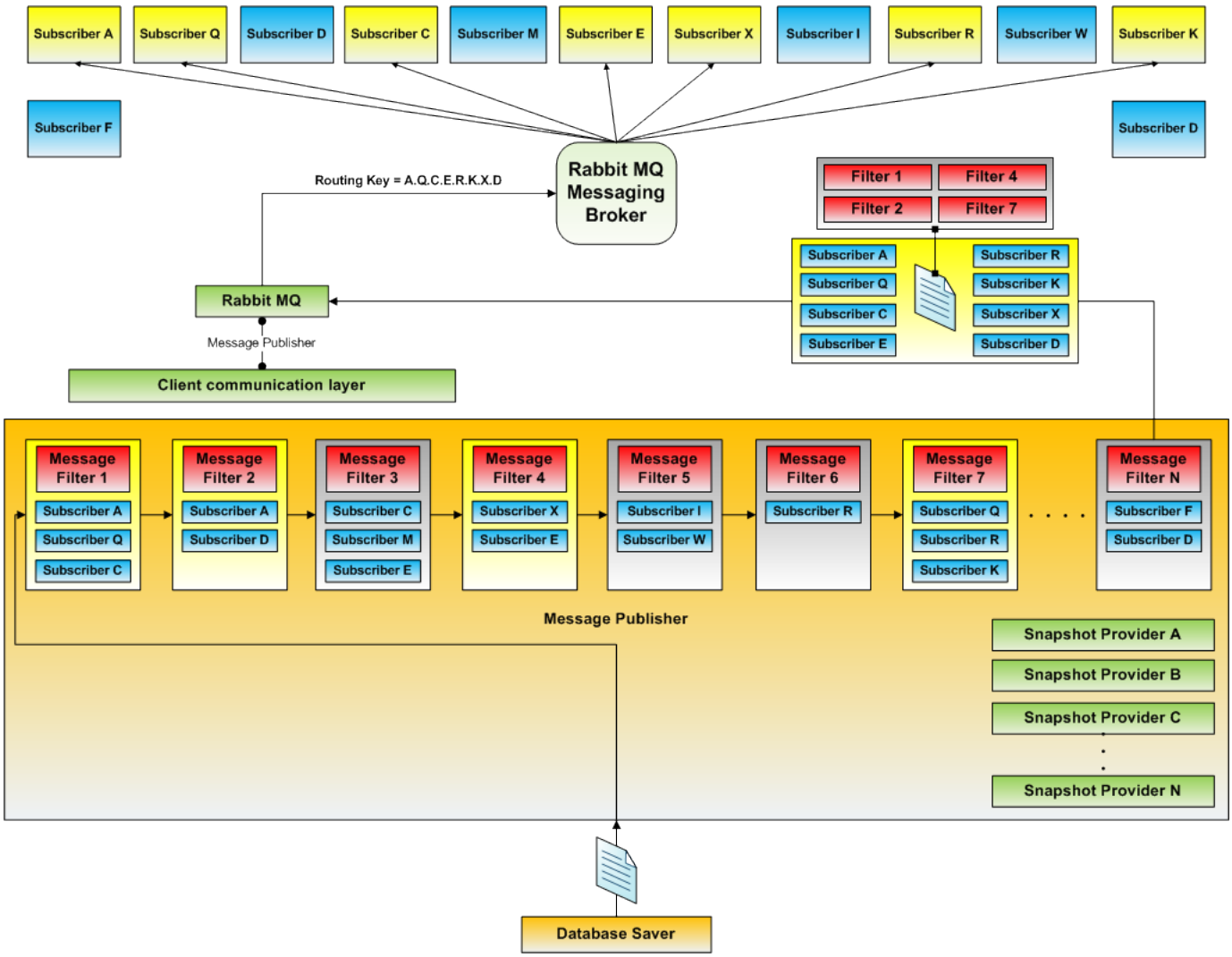


Figure 8.16: Real-time message publishing process

Whenever the `MessagePublisher` receives a message, it loops through all currently stored `MessageFilters` and checks whether they match the message or not. With each matched filter, the `MessagePublisher` stores its identifier into a message, takes a corresponding subscriber identifiers, that are interested in this message and pass them along with the message to a communication message publisher provider. The communication provider constructs from received subscriber identifiers a routing key and with this routing key sends the message to a corresponding message broker, which ensures the correct distribution of the message to interested clients.

8.1.6 Reporting

One of the requirements of monitoring system was the support to automatically create and distribute client confirmations and latency reports. To satisfy this requirement, two report providers, which extend the application framework report manager module, were implemented. The first provider is a confirmation provider, which provides three different basic types of client confirmation:

- **Delimited confirmation** - Confirmation created as a delimited file, where user can define a custom column delimiter.
- **Fixed length confirmation** - Confirmation created as a fixed length file, where user can define for each column its fixed width, the alignment character, which should be used to align the column's content and align mode, which tells the column, whether the content should be centred or aligned to the left or right side.
- **Excel confirmation** - Confirmation created as an excel sheet.

Because each client may require different information, which should be included in a confirmation, it was important to provide the user with functionality to customize the confirmation content. All three above mentioned confirmation types support besides their specific settings the following customizations:

- **Custom trades filter** - User can choose, what trades should be included in the confirmation by selecting the list of clients and markets indicating for whom and where the trades were executed.
- **Custom number of columns, their order and names** - User can choose how many columns with what names and at what order will be added to confirmation and whether the column header should be included or not.
- **Custom column content** - The user can choose what value should be put into each column for each executed trade. It can be a constant value, the value of some tag present in the executed trade or one of predefined parameters such as MIC code of market, where the trade was executed, the name of the instrument, the gross value of executed trade, settlement date, commission etc. User can also choose what characters should be trimmed from the column values, what format should be applied on them and what constant should be used in case the specified tag is not present in the executed trade.
- **Custom file name** - Each confirmation can be named as required by client.

It is important to note here, that in case the above mentioned confirmation types and their customization do not satisfy client's needs, it is still possible to implement new types by creating a new report provider extension, which is a very easy procedure. Once the confirmation template is created its generation can be scheduled by a scheduler manager and distributed afterwards.

The second type of report provider provides the functionality to create a so called summary reports. These reports correspond to the summary views described in Section 8.1.5.4 and therefore its content can be any summarization, which can be defined as a summary view. This means, that it covers the ability to create not just the requested latency reports but also many other interesting reports such as traded volume reports, P&L reports, message count reports etc. Because of the flexibility of summary views the possibilities of what data these reports can contain are very rich.

8.2 Client implementation

Monitoring client was silimar to a server implemented as an external module of application framework and provides the user interface to visualize a trading infrastructure. To get an overview about what components and modules comprise the monitoring client, its architecture is illustrated in Figure 8.17.

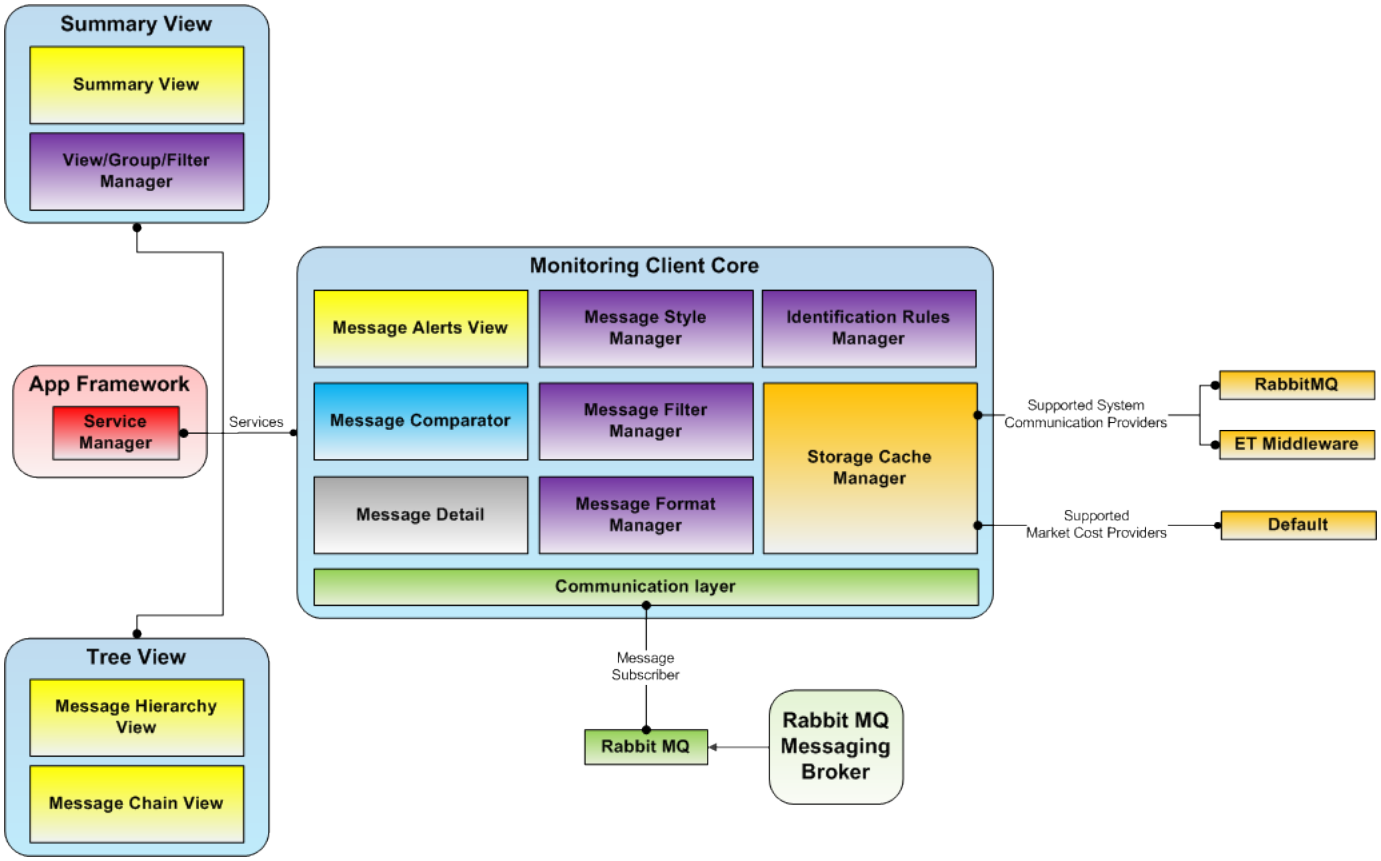


Figure 8.17: Monitoring client architecture

As you can see the client implementation consists of three external modules, the core module, which contains all core components, the summary view module, which is responsible for visualization of summarized data and for management of summary view objects and a tree view module, which implements windows for visualization of message hierarchies and message chains. The summary view and tree view modules are dependent on a core module, so can't be registered without it. Because the implementation of monitoring client is based on an application framework, a lot of important technical details were already described in Section 7.3 and therefore the following sections describe only the most important technical details, which are specific for a monitoring system only. A detailed description of monitoring client from the user point of view can be found in [Postulka10] included on the attached CD.

8.2.1 Communication

The communication layer was implemented to allow client terminals to receive messages published from a monitoring server. Because the message publisher on monitoring server side is developed as an extension to provide a flexibility to change the communication middleware in the future, the message subscriber on client side had to be implemented as an extension of communication layer too. The following paragraphs describe how to implement such message subscriber and how to integrate it into a client's communication layer.

All important communication classes are implemented under a `WAC.ET.Monitor.Client.Core.Communication` namespace and the first instance initialized from it is a `MessageSubscriptionM-`

anager. The main purpose of this class is to discover, load and initialize a message subscriber instance and to manage all client subscriptions and deliver the received messages to interested client components and modules. The registration of message subscriber instance is done through an extension point shown in Example 8.23 where each instance have to implement an IMessageSubscriber interface, which declaration you can see in Example 8.24.

Example 8.23 Message subscriber extension point declaration

```
<ExtensionPoint path = "/Monitor/Communication/MessageSubscriber">
  <ExtensionNode name="MessageSubscriber"
    type="WAC.ET.Monitor.Client.Core.Communication.MessageSubscriberExtensionNode">
    <Description>Registers a message subscriber allowing client to subscribe for real-time flow of ↩
        messages published by server.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

Example 8.24 IMessageSubscriber interface declaration

```
public interface IMessageSubscriber : IComponent
{
  /// <summary>Have to be raised whenever the subscriber receives a new message from server.</summary>
  event MessageReceived OnMessageReceived;

  /// <summary>Identification of the subscriber.</summary>
  string SubscriberId { set; }

  /// <summary>Tells the subscriber to subscribe for messages.</summary>
  /// <param name="interestedSubscribers">The list of interested subscribers.</param>
  void SubscribeForMessages(HashSet<string> interestedSubscribers);

}
```

The interface is quite simple and declares only one method to tell the subscriber to subscribe for messages with given subscriber identifiers. After the component is subscribed for messages then whenever it receives a message from the monitoring server it have to fire an OnMessageReceived event, which is used to pass the received messages to a MessageSubscriptionManager that distributes them to other client components. As you can see in Figure 8.17, the current implementation of monitoring client includes the message subscriber extension, which uses for its communication a RabbitMQ middleware.

8.2.2 GUI real-time updates optimization

Because the whole trading infrastructure may generate tens of messages per a second and these messages may result in hundreds of user interface updates, it was important to optimize the update process of user interface, so the client terminal will not be flooded and will stay responsive. For this purpose a special component called GuiUpdateBuffer was implemented and is used on all client application windows. Its main task is to buffer the user interface updates based on a given configuration and pass them subsequently to the corresponding window. The most important parameters of a GuiUpdateBuffer instance are an *updateInterval* and *maximumNumberOfUpdatesPerBatch* parameters. The *updateInterval* parameter specifies how often should be the buffered updates passed to the registered window and the *maximumNumberOfUpdatesPerBatch* parameter is used to specify the maximum number of updates, which could be passed within the scope of one update event. This optimization ensures that the application's GUI thread will be always responding and doesn't hang in case of many updates within a short time interval.

The other important functionality, which the `GuiUpdateBuffer` component provides, is the ability to mark each update with a unique key, that can be later used to replace this update with a new value. If you think about this kind of optimization, you realize that it can significantly boost the overall client application performance in specific situations. Let's give you an example on a summary view - each summary view is represented by a grid component, which consists of several rows and columns and whenever client receives a new message in which is the summary view interested, it results in a recalculation of at least one grid row. Because it is a common scenario, that hundreds of received messages update the same summary view row, the usage of `GuiUpdateBuffer` reduces the amount of row updates radically. This is possible because each update passed to the `GuiUpdateBuffer` instance is marked with a unique key, that corresponds to a specific grid row, so whenever the summary view receives a large batch of messages within a short time interval, many updates are subsequently replaced in the background and finally only few rows of grid component are updated.

The last but not least optimization process used within the `GuiUpdateBuffer` component is, that the component checks the state of the underlying window at each time it should pass the updates to it and in case the content of the window is not visible at this moment or the whole application is minimalized, the updates are not passed and waits until the window becomes visible again. This can also significantly minimize the application workload.

8.2.3 Core components and basic modules

As already mentioned at the beginning of the chapter, the monitoring client comprise of several core components and modules and the following paragraph briefly describe the most important ones.

- **Storage cache manager** - Used to manage and cache all important structures downloaded from the server, such as client, instrument, system and market definitions. It is also listening for all changes of these structures and updates them accordingly, so the client has always the correct data.
- **Message format manager** - Because each message comprise of many tags, it was quite important to give the user possibility to customize the format, which should be used to describe the message in other client components such message hierarchy view or message chain view. Message formats provide user with choice, which tags and basic properties will be included in the message description of each message type and how they should be integrated in a custom message text. The message format manager then provides centralised management of all these message formats.
- **Message filter manager** - Responsible for centralised management of all message filters. It provides the basic windows and dialogs to allow user to create, edit and delete message filters. These filters can be used later in other client components and modules such as message hierarchy view and gives user the option to quickly filter out the important messages.
- **Message style manager** - Responsible for centralised management of all message styles and lets user to customize the way how will be the messages displayed in other client components such as message hierarchy view or message chain view. Through the message style the user can define what colour and font will be used to visualize different types of messages.
- **Identification rules manager** - Allows user to create new message identification rules or update or delete the existing ones. It provides dialogs and windows for rules management and automatically passes all changes directly to the monitoring server where they are immediately integrated into message classification process.

- **Message detail** - Window used to display details about a given message. Each message is displayed as a list of tags, where user can see, which tags are required for a given type of message and which not. Each tag is also completely described with all tag details as defined in FIX protocol, so the user do not have to search for tag definition and can see it immediately in the message detail window.
- **Message comparator** - Window used to compare two message instances and highlight the differences between them.
- **Message alerts view** - Window containing all important alerts identified during the message processing. These alerts are used to notify user about messages for which the client, instrument or identification rule don't exist, about duplicate messages, messages for which doesn't exist the parent message within the message hierarchy etc.
- **Message hierarchy view** - One of the most important windows, which is used to visualize the message hierarchies through a predefined or dynamic message filters. Besides the hierarchy view, the window supports also a flat view, which is used to display messages in a chronological order. Users can also apply their custom message formats and styles on this window to suit their needs.
- **Message chain view** - Window used to visualize message chains, so the user can see, how the given message flows through the whole trading infrastructure. The window is implemented in a similar way as a message hierarchy view and therefore supports same customizations.
- **Summary view** - Can be considered as a main monitoring window, which gives the user an overview about the messages flow within the whole trading infrastructure. The summarized data can be fully customized and provides information about currently traded volumes, order states, P&L and latencies. It is also used for very quick message filtering and for displaying message hierarchies corresponding to the chosen summary area.
- **Summary view objects manager** - Responsible for centralised management of summary objects, such as summary filters, groups and views, which are then used to create and customize the summaries.

For more detailed description of each above mentioned component and module please see the [Postulka10] included on the attached CD.

8.2.4 Built-in configuration extensions

Besides the core components and modules described in the previous section, the implementation of monitoring client includes also a couple of configuration providers, which were developed in a form of extensions and are used to configure the server side extension components. The current implementation includes three types of them and these are as follows:

- **System communication configuration providers** - Lets user to configure the system communication providers, which are used on the server side to communicate with the systems within the trading infrastructure. All providers have to be registered through an extension point exposed by the `WAC.ET.Monitor.Client.Core` library, which declaration is shown in Example 8.25 and have to implement an `ISystemCommunicationConfigurationProvider` interface described in Example 8.26. The current implementation of monitoring client includes two providers, which corresponds to an ET middleware and RabbitMQ system communication providers implemented on the server side.

Example 8.25 System communication configuration provider extension point declaration

```
<ExtensionPoint path = "/Monitor/Storage/SupportedCommunicationProviders">
  <ExtensionNode name="SystemCommunicationConfigProvider"
    type="WAC.ET.Monitor.Client.Core.Storage.SystemCommunicationConfigurationProviderExtensionNode">
    <Description>Registers a new system communication configuration provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

Example 8.26 ISystemCommunicationConfigurationProvider interface declaration

```
public interface ISystemCommunicationConfigurationProvider
{
  /// <summary>List of supported system communication providers.</summary>
  List<string> SupportedProviders { get; }

  /// <summary>
  ///  Creates the default configuration for a given type of system communication provider.
  /// </summary>
  /// <param name="providerName">Provider for which should be the configuration created.</param>
  /// <returns>Configuration instance with settings of given system communication provider.</returns>
  CommunicationProviderConfiguration CreateDefaultConfiguration(string providerName);

  /// <summary>Initializes the provider.</summary>
  void Initialize();
}
```

- **Market cost configuration providers** - Configuration providers allowing the user to configure separate market cost providers, which are used on the server side during the message processing, concretely by a P&L manager component to calculate online revenues and costs of the electronic trading service. Each market cost provider implemented on the server side should have the corresponding configuration provider implemented on the client side, so the user is able to configure the server component. Same as in previous paragraph, all configuration providers have to be registered through an extension point exposed by the `WAC.ET.Monitor.Client.Core` library. Its declaration is shown in Example 8.27 and each registered provider have to implement an `IMarketCostConfigurationProvider` interface described in Example 8.28. The current implementation of monitoring client includes only one configuration provider, which corresponds to the server side default market cost provider.

Example 8.27 Market cost configuration provider extension point declaration

```
<ExtensionPoint path = "/Monitor/Storage/SupportedMarketCostProviders">
  <ExtensionNode name="MarketCostConfigProvider"
    type="WAC.ET.Monitor.Client.Core.Storage.MarketCostConfigurationProviderExtensionNode">
    <Description>Registers a new market cost configuration provider.</Description>
  </ExtensionNode>
</ExtensionPoint>
```

Example 8.28 IMarketCostConfigurationProvider interface declaration

```
public interface IMarketCostConfigurationProvider
{
  /// <summary>
  /// List of supported market cost providers.
  /// </summary>
  List<string> SupportedProviders { get; }

  /// <summary>
  ///  Creates the default configuration for a given type of market cost provider.
  /// </summary>
  /// <param name="providerName">Provider for which should be the configuration created.</param>
  /// <returns>Configuration instance with settings of given market cost provider.</returns>
  MarketCostConfiguration CreateDefaultConfiguration(string providerName);

  /// <summary>
  /// Initializes the provider.
  /// </summary>
  void Initialize();
}
```

- **Confirmation providers** - Providers used to configure the settings of client confirmations. These settings are then applied by the monitoring server during the confirmation generation process. Contrary to the previous two types of configuration providers, confirmation providers have to be registered through an already known extension point exposed by a reporting manager, which is implemented as a part of application framework. The reason for this is, that the client confirmations are implemented as special types of application reports and therefore each registered provider have to implement an `IReportConfigurationProvider` interface, which declaration is shown in Example 8.29.

Example 8.29 IReportConfigurationProvider interface declaration

```
public interface IReportConfigurationProvider
{
  /// <summary>List of supported confirmation providers.</summary>
  List<string> SupportedReports { get; }

  /// <summary>Initializes the provider.</summary>
  void Initialize();

  /// <summary>Uninitialize the provider.</summary>
  void Uninitialize();

  /// <summary>
  ///  Gets the configuration panel, through which the user can configure the given report.
  /// </summary>
  /// <param name="report">Report to be configured.</param>
  /// <param name="reportConfiguration">Current configuration string of the report.</param>
  /// <returns>Configuration panel used to configure the given report.</returns>
  BaseReportConfigurationPanel GetPanel(string report, string reportConfiguration);
}
```

Chapter 9

Conclusion

During the development of the monitoring system it turned out that the most important part was a thorough analysis of environment in the sponsoring company. It was fundamental to deeply study the problematic of electronic trading, to analyse the existing and future trading infrastructure and to gather as many requirements as possible from all potential users of the system. During the system development it was necessary to frequently consult the solution with representatives of the sponsoring company and thereby I would like to thank all of them for their support and willingness to help especially in the early stages of the system implementation.

9.1 Meeting the targets

The goal of this thesis was to analyze processes in a brokerage firm and based on this analysis to design and implement a system for real-time monitoring of electronic trading infrastructure. In Section 3.3 were described the main system requirements, which had to be taken into account during the system development and the following paragraphs summarize whether the current monitoring system implementation fulfil them or not.

- **Current and future trading infrastructure support** - Monitoring system supports both the obsolete solutions, which are still present in the trading infrastructure and distributed algorithmic engines, which subsequently replace these obsolete solutions. Because the obsolete systems can't be easily customized a monitoring agents were implemented. These agents are responsible for publishing messages from the obsolete system's persistence to a monitoring server and they use for their communication the RabbitMQ middleware. Regarding the algorithmic engines, the message publishing process was implemented directly into them and the communication is based on ET middleware. Because the monitoring system supports different types of system communication providers, which can be registered through server's extension points, it will be very easy to integrate a new type of trading systems in the future.
- **Enterprise-wide infrastructure monitoring** - Because all current types of systems within the trading infrastructure are supported by the monitoring system, all of them can be monitored at real-time. For tracking the messages through the whole trading infrastructure, the special components on both server and client side were implemented. On server side it is a message classification processor, which is responsible for building the message chains and on client side it is a message chain view window, which is used to visualize these chains to the user.

- **High performance** - The sufficient performance was one of the key requirements on which were put a big importance during the development of each component involved in a real-time message processing. Whether it was a selection of communication middleware, implementation of persistence storage, implementation of message processors or implementation of client components used to visualize the data flow. Many optimization techniques such as usage of file buffers for message persistence instead of database persistence, usage of GUI update buffers for optimized data visualization, usage of custom message filter storages for faster message filtering and others were already discussed throughout the text. All these optimizations ensure that the monitoring system is capable of processing of tens of messages per second and hundreds of thousands messages per day with a very low CPU consumption. The only limitation of current implementation is a physical memory available on a machine running the monitoring server. This is because the current implementation keeps all processed messages of a current trading session in memory to ensure the fastest message filtering process. However, because the importance was put also on memory consumption, the memory requirements are not too high for a current message flow and in general the server consumes about 1GB of operation memory per five hundred thousand of messages. Despite this fact an optimized solution was already analysed and is described in the Section 9.3.
- **Comprehensive filtering functions** - The flexibility of message filtering was brought to its maximum. There are two standard message filters implemented, the first is a tag based filter, which allows user to filter messages based on any tag defined in the FIX or internal protocol. For each tag user can choose from different comparison operations and can create complex filtering formulas from them. The second type of message filter is a filter, which filters messages against basic message properties, such as client, market, system, instrument and other identifiers and same as with tag based filters the user can define complex filtering formulas comprising of basic property rules. Even though these two filters are more than sufficient for all types of monitoring purposes, the monitoring system can be extended by new types of message filters, which gives the system another level of flexibility for the future.
- **Error alerting** - During the real-time message processing each message is validated against set of predefined rules and whenever an error is detected, the message is marked with a corresponding error type. The predefined rules check whether the client and instrument exists in the external storage, whether there exists an identification rule, which matches the message, whether the message is duplicate or not, whether the parent message in message hierarchy exists, whether the instrument identification can be retrieved from the message etc. All messages, which do not pass the validation process are then displayed to user in the client terminal in a message alerts view window.
- **Customizable order state overview** - Because the orders state overview was implemented as a part of a summarization process its customization is very flexible. Each user can create his own summary view, which will completely suit his needs and because the summary view supports all important order states there is no limitation in visualizing.
- **P&L calculation** - Real-time P&L calculation is implemented also as a part of summarization process and therefore there is no limitation in its customization. Besides the custom grouping the user can choose in what currency should be the revenues and costs displayed and what types of costs should be included in the summary view. Each custom view then can be used to generate P&L reports through a summary report provider and distribute them with one of the supported providers. Because the calculation process of market costs is not trivial and because each market can have different rules for fee calculation, the monitoring system supports registration of new types of market cost providers as a system extension.
- **Customizable interface** - User interface of client application was developed in an extremely flexible way, so the users can tailor the interface to their specific needs and tastes. Users can create complex application

layouts between which they can switch on the fly, most of the application windows can be configured for user's needs, users can customize context menus, key shortcuts, action events, binding between different types of windows, message styles, message formats etc.

- **System configuration** - All important parts of monitoring system can be configured directly from the client application. It is possible to configure with few clicks a new system, which should be monitored, to choose and configure the corresponding communication provider, add a new market and configure its corresponding market cost provider, add new system user and copy existing layouts and window configurations from other already existing users etc.
- **Reporting** - Reporting module was implemented as an extensible component, which can be extended by new types of report providers and new types of report publishers. The current implementation includes extensions, which cover all required functionality. From report providers it is a client confirmation provider, which is used to generate customizable confirmations and summary report provider, which allows to generate reports based on customizable summary views, which cover also P&L and latency reports. Currently supported publishers are FTP, email and file system publisher. The report generation and distribution process then can be scheduled through a task scheduler module, which allows user to define and configure different jobs and triggers.

From above paragraphs it is quite clear, that the current implementation of monitoring system not only fulfills all main system requirements, but in many cases exceeds them. Besides that, the exceptionally modular design of whole system provides a very easy way to extend its functionality in the future.

9.2 Deployment in the sponsoring company

The system is fully deployed in the test environment for two months and proves its robustness, very easy usage and rich set of monitoring functions. In test environment the system monitors an electronic trading infrastructure, which consists of two FIX based routing systems and eight instances of algorithmic engines, where for monitoring of routing systems are used monitoring agents. The monitoring system has no problems with processing hundreds of thousands of messages per day without any sign of delays and still provides the fast filtering functions. Because the system proves its stability in test environment, it was deployed into production two weeks ago as well, although it monitors only routing systems of trading infrastructure yet. Until now there was no issue in the production environment either, so it is very likely that the rest of production trading systems will be added and monitored in the near future, too.

9.3 Future enhancements

Even though the current implementation of the system meets all important targets of sponsoring company and suits the major of users needs, its functionality can be still enhanced and a lot of new features can be added. However, because the monitoring system is a very complex project it was not possible to include all these improvements into the current implementation, because they would be out of the scope of this thesis. Despite this fact, the following sections briefly describe the main concept of the most interesting improvements, which will be probably implemented in the near future by a sponsoring company.

9.3.1 History support

Because there are situations when users need to search messages from a previous trading sessions, e.g. because of compliance reasons, it would be nice to support historical queries. This fact was already taken into account during the system development and therefore the whole electronic trading traffic is persisted into a database, so the messages can be retrieved and processed in the future. The current implementation of database persistence is already briefly described in Section 8.1.1 and the only part missing to fully support the historical views is a set of dynamic queries, which will be used to transform message filters into SQL statements, that will return the requested messages and transform them into same objects as used during the real-time message processing. This two-side transformation mechanism will allow simulating historical queries with same application processes as during the real-time message processing and provides user the transparent application interface. The user then can use his predefined message filters and only specifies a date range on which should be these message filters applied.

9.3.2 User roles and permissions

The monitoring system should be used by a wide range of users from different departments and therefore a comprehensive permission management would be necessary in the future. For now it is not crucial, that the system lacks of this functionality because it is used especially by well informed users, but it have to be implemented in a very near future to prevent general users from using application functions, which are not designed for their needs. The analysis of user roles and permissions was already done and it turned out that the most flexible solution would be an implementation based on access control lists. Current database schema already contains tables, which will be used for this purpose and application framework contains components, which are used for validation if the user can use a specific type of action event in the current application state. This means that the integration of user permissions will be very easy task and will consist of an implementation of components on both server and client side for user roles and permissions management and from adding the corresponding validation calls into application framework components.

9.3.3 Market data integration

In the beginning of Chapter 7 was described that one of the reasons to implement an application framework was an easy integration of at first look independent applications. The market data integration represents just this type of integration where an independent application called Market Data Monitor, which is developed separately in a sponsoring company as one of the external modules of application framework, will be integrated together with a monitoring system. Because both systems are developed as application framework external modules, they can be used together right now without any modifications within the same application interface. Advantage of such solution is that the user is using for both modules the same application with common look and control. Besides that the user can create application layouts including both market data specific windows and trading infrastructure monitoring windows. The next step in integration of these two modules will be an interconnection of some of their functions, e.g. very interesting and useful function will be to allow to bind market depth window displaying the current bids and asks on a given stock to a message hierarchy window. This allows the user to immediately see, what was the exact market situation on a given instrument, when the client's order was received. This type of information is very often important for client support desk. It should be quite easy task to allow such binding between windows of separate modules. Programmers of both modules just extend the list of supported binding events on corresponding windows and implement publishing and reception of these events on them.

9.3.4 Memory storage optimization

In Section 9.1 it was described that the only important limitation of the monitoring system is its memory usage, because it keeps all processed messages in the memory and use them for fast message filtering. Even though this is not so relevant for current message flows it can become a bottleneck with the growing number of messages. Right now the monitor server requires about 1GB of physical memory for five hundred thousand of messages, which is not too much with today's hardware, but it would be nice if this limitation is removed and the system become less resource-consuming. The solution for memory usage reduction was also already analysed but not very deeply, so the following text should be taken as a draft only.

The whole solution is inspired in memory management of typical operating system – paging scheme. Global message storage in this solution represents the whole virtual address space and each message represents a memory page. Just like memory page, every message may be stored in the system memory, secondary memory or in both memories. When the system needs to work with the message which is not in the system memory ("page fault") the message must be loaded into system memory from the secondary memory. Messages can be stored into secondary memory as a string and each of them can be compressed to achieve faster storing time. In typical operating system algorithm like LRU or LFU is used when page should be replaced. These algorithms are based on simple idea: page which has not been accessed / modified for a long time won't be accessed / modified in near future either. The monitoring system should use a different algorithm based on state of order, to which are the messages related and it will flush to secondary memory only those messages, which are related already closed (e.g. fully filled, rejected, expired etc.) orders.

Open and active orders cover only a limited subset of all messages and therefore the memory storage could be split into two parts: one small part (like cache of the processor) is still in system memory while the other part is stored in the secondary memory (file). If you think about this solution, this splitting will not affect a real-time message processing at all, because all monitoring server components involved in real-time message processing work with messages related to open orders only. The fact that the order is closed ensures that there will be no other message related to this order and therefore the order can be stored into a secondary memory and deleted from the main system memory.

The only time when the monitoring system needs to retrieve messages related to already closed orders is whenever some of the server side components such as reporting provider or client components such message hierarchy view, message chain view etc. explicitly demand them through message filters. The filtering process may be with this solution a little bit slower, because the corresponding messages have to be loaded from the secondary storage, but if this loading mechanism is implemented efficiently, it should not affect the performance too much. For fast searching of messages in file(s) there can be used indexing mechanism telling the server, in which file and at what position is the message stored. All in all it seems that the main concept of primary and secondary memory storage is a right step to significantly reduce the memory consumption of monitoring server. Yet, but it has to be analysed more thoroughly before its implementation.

Appendix A

Content of attached CD

An integral part of the thesis is a CD disc containing the actual thesis in its electronic form, system's user manual, source files, installation files required for correct running of the system and an simulation environment allowing reader to launch and evaluate the system. In almost every directory on the CD is the **readme.txt** file that describes the content of the directory in detail, so it can be used to obtain accurate information.

Content of attached CD:

- `documentation` - Thesis and user manual both in electronic form.
- `install` - Installation files and scripts required for correct running of the system.
- `simulation` - Simulation environment allowing simulation of simple trading infrastructure and presentation of monitoring system functionality.
- `sources` - Complete source files of both application framework and monitoring system, third-party libraries and required libraries developed in sponsoring company

Appendix B

Glossary

These are the meanings assigned to terms in this work. Most of them are taken from referenced publications. Terms in italics in definitions denote a cross reference.

Access

The possibility to participate in a market.

Back office

Organisational unit responsible for post-trade activities such as *clearing* and *settlement*.

Best execution

The obligation of broker/dealers, and others to *execute* customer orders at the best price currently available.

Bid

The highest price any buyer is willing to pay for a given security at a given time.

Broker

Firm which operates in a market on behalf of other participants to arrange transactions without being a party to the transactions itself.

Broker-dealer

Any individual or firm in the business of buying and selling securities for itself and others.

Care order

Order is electronically routed to the trading terminal where usually sits a sales trader, who is responsible for further order processing.

Clearing

The process of transmitting, reconciling and sometimes *confirming* instructions to transfer instruments prior to *settlement*.

Commission

A fee charged by a *broker* for his/her service in facilitating a transaction, such as the buying or selling of securities.

Confirmation

The written statement acknowledging a securities transaction. More generally, any formal communication which reiterates or verifies an agreement.

Crossing engine

Alternative trading system that matches buy and sell orders electronically for *execution* without first routing the order to an exchange. The advantage of the *crossing engine* is the ability to execute a large block order without impacting the public *quote* and reduction of trade costs, because there is no market cost charged for executed trades.

Derivative

A financial instrument whose characteristics and value depend upon the characteristics and value of an underlying, for example *stock*.

Discount broker

A *broker* which *executes* buy and sell orders at *commission* rates lower than a *full-service brokerage*, but which typically provides fewer services such as research and advice.

DMA

Direct market access: a system of trading that allows buy-side firms to have easier access to *liquidity*, and allows them more control over their trades. Benefits of DMA include that less work is required to be done by the *broker*. Orders are directly routed to the exchange, so no *broker* intervention is needed, and transactions costs (such as *commissions*) are lower for the firm.

Electronic order routing

Type of electronic delivery of orders to *execution* system.

ET

Electronic trading. In broad terms, this refers to any use of electronic means of sending orders (*bids* and *offers*) to the market, *electronic order routing*, automated centralised execution and subsequent dissemination of price and volume information. See Section 1.1 for more details.

Execution

The matching of orders or trade proposals which turns them into actual trades.

Exchange

Any organization, association or group which provides or maintains a marketplace where securities can be traded.

Floor trader

An *exchange* member who executes orders on the floor for his/her own account.

Front office

Organisational unit that is "client facing". It is responsible for fulfilment client orders and consists mainly of sales staff and traders.

Full-service broker

A *broker* which, in addition to *executing* trades for its clients, also provides them with research and advice. Significantly more expensive than discount *brokers*, which only *execute* trades.

Liquid (market)

Three aspects of liquidity are *tightness*, *depth* and *resiliency*. It is characterised by the ability to transact in a market without markedly moving prices.

Market data

Refers to quote and trade related-data associated with equities and other investment instruments. Includes various pricing information (such as the most recently traded price), and various volume information (such as the number of contracts that were most recently traded).

Market depth

Amount of outstanding orders pending (possibly at different prices) on either side of the market.

Middle office

Organisational unit responsible for position-keeping, risk management, trade *confirmation* and *P&L* calculations. Usually it is also a mediator between *front office* and *back office*.

Offer

The lowest price that any investor has declared that he/she will sell a given security for.

Open outcry

Method of public auction in which verbal *bids* and *offers* are made in the trading rings of *exchange*.

P&L

Profit and loss statement.

Quote

The highest *bid*or lowest *offer* price available on a security at any given time.

Resilient (market)

Market which continues to function in an efficient and *liquid* manner at times of great price uncertainty and market stress.

Retail investor

An individual who purchases small amounts of securities for him/herself.

Settlement

Completion of a transaction by exchange of instrument.

Smart order routing

Service that identifies the market currently *quoting* the best price for given order and automatically route orders accordingly.

Stock

An instrument that signifies an ownership position (called equity) in a corporation, and represents a claim on its proportional share in the corporation's assets and profits.

Stock broker

Broker who deals primarily with transactions involving *stock*

Stock issuer

A company offering (or having already offered) securities for sale to investors.

Stock market

General term for the organized trading of *stocks* through *exchanges* and other marketplaces.

STP

Straight-through processing: the capture of trade details directly from *front office* systems to *middle office* and *back office*. Completes automated processing of confirmations and settlement instructions without the need for rekeying or reformatting data.

Tightness (market)

A measure of *liquidity* derived from the difference between buying and selling *quotes*.

Transparency

Ability of market participants to observe (pre-trade) quotes, (post-trade) prices and volumes in a timely fashion.

TWAP

Time Weighted Average Price. It is the average price of contracts or shares over a specified time.

VWAP

Volume Weighted Average Price. A measure of the price at which the majority of a given day's trading in a given security took place. Calculated by taking the weighted average of the prices of each trade.

Appendix C

Bibliography

[AMQPWG09] *Advanced Message Queuing Protocol*, http://www.amqp.org, .

[AMQPWG09a] *AMQP: A General-Purpose Middleware Standard*, http://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1, .

[ASF08] *Apache ActiveMQ*, http://activemq.apache.org/, .

[ASF09] *The Apache Software Foundation*, http://www.apache.org/, .

[ASF09b] *Apache Qpid*, http://cwiki.apache.org/confluence/display/qpid/Index.

[Acrux10] *Acrux Threading.NET*, 2010, Copyright © 2007-2008 Acrux Software Pty Ltd, http://www.acruxsoftware.net/.

[B2BITS09] *B2BITS, the EPAM Capital Markets Competency Center*, http://www.b2bits.com/, .

[Cgfs01] Working group established by the Committee on the Global Financial System of the central banks of the Group of Ten countries, *The implications of electronic trading in financial markets*, January 2001, 92-9131-613-X, Bank for International Settlements, Basel, Switzerland, Copyright © 2001 Bank for International Settlements, http://www.bis.org/publ/cgfs16.pdf.

[CohesiveFT09] *Cohesive Flexible Technologies*, http://www.cohesiveft.com/, .

[CompFact10] *Component Factory*, 2010, Copyright © 2010 Component Factory Pty Ltd, http://www.componentfactory.com.

[DDS09] *Data Distribution Service*, http://en.wikipedia.org/wiki/Data_Distribution_Service, .

[DockPanel10] *DockPanel Suite*, 2010, http://sourceforge.net/projects/dockpanelsuite/.

[ELCA09] *ELCA*, 2009, http://www.elca.ch/, Copyright © 2009 ELCA.

[ETH03] *Programming Languages and Runtime Systems Research Group - ETH Zentrum, Zürich*, August 2003, http://www.cs.inf.ethz.ch/gutknecht/index.html.

[Ericsson09] *Ericsson*, http://www.ericsson.com/, .

[Ericsson09a] *Erlang*, http://www.erlang.org/.

[Et09] *Electronic Trading*, March 2009, http://en.wikipedia.org/wiki/Electronic_trading.

[FIXFlyer09] *FIX Flyer*, http://www.fixflyer.com/, .

[FIXFlyer09a] *Daytona: Trade Monitoring Framework and Support Tools*, http://www.fixflyer.com/html-files/daytona.html, .

[FIXProtocolLtd01] *FINANCIAL INFORMATION EXCHANGE PROTOCOL (FIX) Version 4.2 with Errata 20010501*, FIX Protocol Limited, Copyright © 2001 FIX Protocol Limited, May 2001.

[FIXProtocolLtd03] *FIX Protocol Implementation Guide First Edition*, FIX Protocol Limited, October 2003, http://fixprotocol.org/implementation-guide/.

[FIXProtocolLtd09] *The FIX Protocol Organization*, http://fixprotocol.org.

[Fiorano09] *Fiorano Software Technologies P Ltd*, http://www.fiorano.com/, .

[Fiorano09a] *FioranoMQ*, http://www.fiorano.com/products/fmq/products_fioranofmq.php, .

[Fowler02] Martin Fowler, Rice, Foemmel, Hieatt, Mee, and Stafford, *Patterns of Enterprise Application Architecture*, Addison Wesley, November 2002, 0-321-12742-0.

[Greenline09] *Greenline Financial Technologies*, http://www.greenlinetech.com/, .

[Henning04] Henning, *A New Approach to Object-Oriented Middleware*, January 2004, http://www.zeroc.com/documents/ieee.pdf, IEEE Computer Society.

[Henning06] Henning, *The Rise and Fall of CORBA*, June 2006, http://delivery.acm.org/10.1145/1150000/1142044/p28-henning.pdf, Copyright © 2009 Association for Computing Machinery.

[Henning09] Henning and Spruiell, *Distributed Programming with Ice (Revision 3.3.1)*, March 2009, http://www.zeroc.com/download/Ice/3.3/Ice-3.3.1.pdf, Copyright © 2003-2009 ZeroC.

[IIOP.NET04] *IIOP.NET project*, 2004, http://iiop-net.sourceforge.net/, Copyright © 2003-2004 ELCA.

[IKVM09] *IKVM.NET*, 2009, Copyright © 2009 Jeroen Frijters, http://www.ikvm.net/.

[JPMorgan09] *JPMorgan Chase & Co.*, http://www.jpmorganchase.com, .

[LShift09] *LShift Ltd.*, http://www.lshift.net/, .

[Mef09] *Managed Extensibility Framework*, 2009, Copyright © 2009 Microsoft Corporation, http://mef.codeplex.com/.

[Microsoft06] *London Stock Exchange Cuts Information Dissemination Time from 30 to 2 Milliseconds*, Microsoft, October 2006, http://switch.atdmt.com/action/FY07_Linux_LSE_Download.

[Microsoft09] *[MS-NRTP]: .NET Remoting: Core Protocol Specification (v20090521)*, May 2009, http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-NRTP%5D.pdf, Copyright © 2009 Microsoft Corporation.

[Microsoft09a] *.NET Remoting Overview*, 2009, http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx, Copyright © 2009 Microsoft Corporation.

[Microsoft09b] *System.Addin Namespace*, 2009, Copyright © 2009 Microsoft Corporation, http://msdn.microsoft.com/en-us/library/gg145020.aspx.

[MonoAddins09] *Mono.Addins*, 2009, http://www.mono-project.com/Mono.Addins.

[MonoAddins09a] *Mono.Addins Documentation*, 2009, http://monoaddins.codeplex.com/documentation.

[MonoDev09] *MonoDevelop*, 2009, http://monodevelop.com/.

[MultiTier09] *Multi-tier architecture*, June 2009, http://en.wikipedia.org/wiki/Multitier_architecture.

[NYFIX09] *TradeScope*, http://nyfix.com/buyside-solutions/messaging-monitoring/tradescope, .

[NYFIX09a] *NYFIX Millennium*, http://nyfix.com/, .

[NetScout09] *NetScout Systems*, http://www.netscout.com/, .

[NetScout09a] *Sniffer Financial Intelligence*, http://www.netscout.com/solutions/fix_protocol.asp#documents, .

[OCI09] *Object Computing, Inc.*, 2009, http://www.ociweb.com/.

[OMG07] *Data Distribution Service for Real-time Systems Version 1.2 (OMG Available Specification formal/07-01-01)*, http://www.omg.org/cgi-bin/doc?formal/07-01-01.pdf, .

[OMG08] *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*, Jan 2008, http://www.omg.org/docs/formal/08-01-04.pdf, Copyright © 1997-2009 Object Management Group.

[OMG09] *The Object Management Group (OMG)*, June 2009, http://www.omg.org/, Copyright © 1997-2009 Object Management Group.

[OTP09] *Open Telecom Platform*, http://en.wikipedia.org/wiki/Open_Telecom_Platform.

[OnixS09] *Onix Solutions*, http://www.onixs.biz/, .

[Orc09] *CameronFIX Universal Server*, http://www.orcsoftware.com/Solutions/Orc-Connect/CameronFIX-Universal-Server/.

[Postulka10] Petr Postulka, *Monitoring system user manual*, November 2010.

[Postulka10a] Petr Postulka, *Monitoring system API documentation*, November 2010.

[PrismTech09] *PrismTech*, 2009, http://www.prismtech.com/, Copyright © 1995-2009 PrismTech Ltd..

[PrismTech09a] *PrismTech Corporation*, http://www.opensplice.com/, .

[Puder05] Puder, Römer, and Pilhofer, *Distributed Systems Architecture: A Middleware Approach*, Morgan Kaufmann, October 2005, 1558606483.

[Quartz10] *Quartz.NET - Enterprise Job Scheduler for .NET Platform*, 2010, http://quartznet.sourceforge.net/index.html.

[QuickFix09] *QuickFIX Engine*, http://www.quickfixengine.org.

[RPC09] *Remote Procedure Call*, http://wiki.answers.com/Q/What_is_RPC_and_RMI_Give_appropriate

[RTI09] *Real-Time Innovations*, http://www.rti.com/, .

[RabbitMQ09] *RabbitMQ*, http://www.fiorano.com/products/fmq/products_fioranofmq.php, .

[Rao95] Rao, *Making the Most of Middleware*, September 1995, Data Communications International 24.

[RedHat08] *Welcome to AMQP, Microsoft*, http://press.redhat.com/2008/10/24/welcome-to-amqp-microsoft/, .

[Rolstadas00] Rolstadas, *Enterprise Modeling: Improving Global Industrial Competitiveness*, July 2000, Springer.

[SEI08] The Software Engineering Institute, *Software Technology Roadmap*, July 2008, http://www.sei.cmu.edu/str/str.pdf, Copyright © 2008 Carnegie Mellon University.

[Satanek09] Jaromir Satanek, *Stroj pro algoritmické obchodování*, April 2009.

[Sb09] *Stock Broker*, April 2009, http://en.wikipedia.org/wiki/Stock_broker.

[SharpDev09] *The Open Source Development Environment for .NET*, 2009, Copyright © 2009 IC#Code, http://www.icsharpcode.net/opensource/sd/.

[Sun00] *Java Remote Method Invocation*, 2000, http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTO
Copyright © 1997-1999 Sun Microsystems.

[Sun02] *Java Message Service (JMS)*, http://java.sun.com/products/jms/, Sun Microsystems, Inc..

[Sun09] *Basic JMS API Concepts*, http://docs.sun.com/app/docs/doc/819-3669/bncdx?l=sv&a=view, Sun Microsystems, Inc..

[Sutton00] Sutton, *Session 1: Middleware Selection*, Copyright © 2001 Springer-Verlag, July 2000.

[TG09] *Thales Group*, http://www.thalesgroup.com/, .

[TIBCO09] *TIBCO Software Inc*, http://www.tibco.com/, .

[TIBCO09a] *TIBCO Rendezvous datasheet*, http://www.tibco.com/multimedia/ds-rendezvous_tcm8-826.pdf, .

[TIBCO09b] *TIBCO Rendezvous*, http://www.tibco.com/software/messaging/rendezvous/default.jsp, .

[Wustl07] *Real-time CORBA with TAO (The ACE ORB)*, June 2007, http://www.cs.wustl.edu/~schmidt/TAO.html.

[Wustl07a] *The ADAPTIVE Communication Environment (ACE)*, June 2007, http://www.cs.wustl.edu/~schmidt/ACE.html.

[ZeroC09] *ZeroC*, 2009, http://www.zeroc.com/, Copyright © 2009 ZeroC.

[iMatix09] *iMatix Corporation*, http://www.imatix.com/, .

[iMatix09a] *OpenAMQ*, http://www.openamq.org/, .